A Crash-Course in Scala

"Scala — Slowly compiled academic language"
— a joke(?) found on Twitter

"Life is too short for `malloc`."
— said Neal Ford at Oscon'13 ☞

Introduction

Scala is a programming language that combines functional and object-oriented programming-styles. It has received quite a bit of attention in the last ten or so years. One reason for this attention is that, like the Java programming language, Scala compiles to the Java Virtual Machine (JVM) and therefore Scala programs can run under MacOSX, Linux and Windows. Because of this it has also access to the myriads of Java libraries. Unlike Java, however, Scala often allows programmers to write very concise and elegant code. Some therefore say "Scala is the better Java".[1]

A number of companies—the Guardian, Duolingo, Coursera, FourSquare, Netflix, LinkedIn, ITV, Disney to name a few—either use Scala exclusively in production code, or at least to some substantial degree. Scala seems also useful in job-interviews (especially in data science) according to this anecdotal report

`http://techcrunch.com/2016/06/14/scala-is-the-new-golden-child`

The official Scala web-page is here:

`http://www.scala-lang.org`

⚠ For PEP, make sure you are using the version 3(!) of Scala. This is the version I am going to use in the lectures and in the coursework. This can be any version of Scala 3.X where $X = \{4, 5, 6, 7\}$. Also the minor number does not matter. Note that this will be the second year I am using this newer version of Scala – some hiccups can still happen. Apologies in advance!

If you are interested, there are also experimental backends for Scala for generating JavaScript code (`https://www.scala-js.org`), and there is work under way to have a native Scala compiler generating X86-code (`http://www.scala-native.org`). There are also some tricks for Scala programs to run as a native GraalVM ☞ image. Though be warned these backends are still rather beta or even alpha. But for the coursework we are going to use the "vanilla" Scala compiler.

[1]from `https://www.slideshare.net/maximnovak/joy-of-scala`, though this might be outdated as latest versions of Java are catching up somewhat

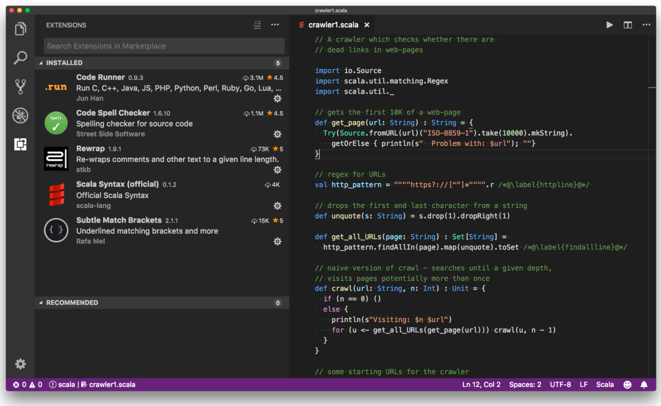


Figure 1: My installation of VS Code / Codium includes the package **Scala Syntax (official)** 0.5.7 from Marketplace. I have also bound the keys Ctrl Ret to the action "Run-Selected-Text-In-Active-Terminal" in order to quickly evaluate small code snippets in the Scala REPL. I use Codium's internal terminal to run `scala` version 1.0.5 which uses Scala 3.7.3.

VS Code and Scala

I found a convenient IDE for writing Scala programs is Microsoft's *Visual Studio Code* (VS Code) which runs under MacOSX, Linux and obviously Windows.[2] It can be downloaded for free from

https://code.visualstudio.com

and should already come pre-installed in the Department (together with the Scala compiler). Being a project that just started in 2015, VS Code is relatively new and therefore far from perfect. However it includes a *Marketplace* from which a multitude of extensions can be downloaded that make editing and running Scala code a little easier (see Figure 1 for my setup).

Actually last year I switched to VS Codium as IDE for writing Scala programs. VS Codium is VS Code minus all the telemetry data that is normally sent to Microsoft. Apart from the telemetry (and Copilot, which you are not supposed to use anyway), it works pretty much the same way as the original but is driven by a dedicated community, rather than a big company. You can download VS Codium from

https://vscodium.com

[2]…unlike *Microsoft Visual Studio*—note the minuscule difference in the name—which is a heavy-duty, Windows-only IDE…jeez, with all their money could they not have come up with a completely different name for a complete different project? For the pedantic, Microsoft Visual Studio is an IDE, whereas Visual Studio Code is considered to be a *source code editor*. Anybody out there knows what the difference is?

What I like most about VS Code/Codium is that it provides easy access to any Scala REPL. But if you prefer another editor for coding, it is also painless to work with Scala completely on the command line (as you might do with `g++` in the second part of PEP). For the lazybones among us, there are even online editors and environments for developing and running Scala programs: for example *Scastie* is one of them. It requires zero setup (assuming you have a browser handy). You can access it at

> https://scastie.scala-lang.org

But you should be careful if you use them for your coursework: they are meant to play around, not really for serious work. Therefore make sure `scala` works on your own machine ASAP!

As one might expect, Scala can be used with the heavy-duty IDEs Eclipse and IntelliJ. For example IntelliJ includes plugins for Scala

> https://scalacenter.github.io/bloop/docs/ides/intellij

BUT, I do **not** recommend the usage of either Eclipse or IntelliJ for PEP: for the small programs that we will write in this module, these IDEs seem to make your life harder, rather than easier. They are really meant to be used when you have a million-lines codebase instead of our small "toy-programs"...for example why on earth am I required to create a completely new project with several subdirectories when I just want to try out 20-lines of Scala code? Your mileage may vary though. `;o)`

Why Functional Programming?

Before we go on, let me explain a bit more why we want to inflict upon you another programming language. You hopefully have mastered Java and soon will master C++ as well, you possibly know Python already... the world should be your oyster, no? Well, as usual matters are not as simple as one might wish. We do require Scala in PEP, but actually we do not religiously care whether you learn Scala—after all it is just a programming language (albeit a nifty one IMHO). What we do care about is that you learn about *functional programming*. Scala is just the vehicle for that. Still, you need to learn Scala well enough to get good marks in PEP, but functional programming could perhaps equally be taught with Haskell, F#, SML, Ocaml, Kotlin, Clojure, Scheme, Elm and many other functional programming languages.

Very likely writing programs in a functional programming language is quite different from what you are used to in your study so far. It might even be totally alien to you. The reason is that functional programming seems to go against the core principles of *imperative programming* (which is what you do in Java and C/C++). The main idea of imperative programming is that you have some form of *state* in your program and you continuously change this state by issuing some commands—for example for updating a field in an array or for adding one to a variable stored in memory and so on. The classic example for this style of programming is a `for`-loop in say Java and C/C++. Consider the snippet:

```
for (int i = 10; i < 20; i++) {
      //...do something with i...
}
```

Here the integer variable `i` embodies part of the state of the program, which is first set to `10` and then increased by one in each loop-iteration until it reaches `20` at which point the loop exits. When this code is compiled and actually runs, there will be some dedicated space reserved for `i` in memory. This space of typically 32 bits contains `i`'s current value…`10` at the beginning, and then the content will be overwritten with new content in every iteration. The main point here is that this kind of updating, or overwriting, of memory is 25.806…or **THE ROOT OF ALL EVIL**!!

…Well, it is perfectly benign if you have a sequential program that gets run instruction by instruction...nicely one after another. This kind of running code uses a single core of your CPU and goes as fast as your CPU frequency, also called clock-speed, allows. The problem is that this clock-speed has not much increased over the past decade and no dramatic increases are predicted for any time soon. So you are a bit stuck. This is unlike previous generations of developers who could rely upon the fact that approximately every 2 years their code would run twice as fast because the clock-speed of their CPUs got twice as fast.

Unfortunately this does not happen any more nowadays. To get you out of this dreadful situation, CPU producers pile more and more cores into CPUs in order to make them more powerful and potentially make software faster. The task for you as developer is to take somehow advantage of these cores by running as much of your code as possible in parallel on as many cores you have available (typically 4-8 or even more in modern laptops and sometimes much more on high-end machines—and we conveniently ignore here how many cores are on modern GPUs, which can be hundreds or even thousands). In this situation *mutable* variables like `i` in the for-loop above are evil, or at least a major nuisance: Because if you want to distribute some of the loop-iterations over several cores that are currently idle in your system, you need to be extremely careful about who can read and overwrite the variable `i`.[3] Especially the writing operation is critical because you do not want that conflicting writes mess about with `i`. Take my word: an untold amount of misery has

[3] If you are of the mistaken belief that nothing nasty can happen to `i` inside the `for`-loop, then you will have to be extra careful with the C++ material.

arisen from this problem. The catch is that if you try to solve this problem in languages like C/C++ or Java, and be as defensive as possible about reads and writes to `i`, then you need to synchronise access to it. The result is that very often your program waits more than it runs, thereby defeating the point of trying to run the program in parallel in the first place. If you are less defensive, then usually all hell breaks loose by seemingly obtaining random results. And forget the idea of being able to debug such code. If you want to watch a 5-minute video of horror stories, feel free to follow … 👉 🤣 (I love the fact, he says at 4:02 mins that he does not understand how the JVM really works… I always assumed I am the only idiot who does not understand how threads work on the JVM. Apparently not. But the point is that I am a functional programmer: I do not care – I do not have to understand them.)

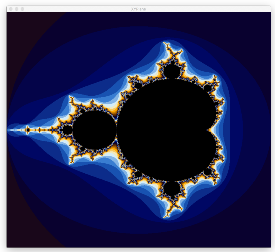
The central idea of functional programming is to eliminate any state and all mutable variables from programs—or at least from the "interesting bits" of the programs. Because then it is easy to parallelise the resulting programs: if you do not have any state, then once created, all memory content stays unchanged and reads to such memory are absolutely safe without the need of any synchronisation. An example is given in Figure 2 where in the absence of the annoying state, Scala makes it very easy to calculate the Mandelbrot set on as many cores of your CPU as possible. Why is it so easy in this example? Because each pixel in the Mandelbrot set can be calculated independently and the calculation does not need to update any variable. It is so easy in fact that going from the sequential version of the Mandelbrot program to the parallel version can be achieved by adding just eight characters—in two places you have to add `.par`. Try the same in C/C++ or Java!

But remember this easy parallelisation of code requires that we have no state in our programs…that is *no* counters like `i` in `for`-loops. You might then ask, how do I write loops without such counters? Well, teaching you that this is possible is one of the main points of the Scala-part in PEP. I can assure you it *is* possible, but you have to get your head around it. Once you have mastered this, it will be fun to have no state in your programs (a side product is that it much easier to debug state-less code and it is also more often than not easier to understand). So have fun with Scala![4]

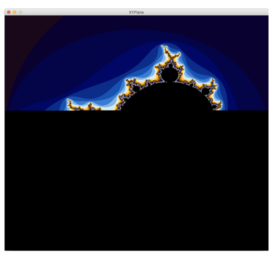
If you need any after-work distractions, you might have fun reading the fol-

[4] If you are still not convinced about the function programming "thing", there are a few more arguments: a lot of research in programming languages happens to take place in functional programming languages. This has resulted in ultra-useful features such as pattern-matching, strong type-systems, laziness, implicits, algebraic datatypes to name a few. Imperative languages seem to often lag behind in adopting them: I know, for example, that Java will at some point in the future support pattern-matching, which has been used for example in SML for at least 40(!) years. See `https://openjdk.org/projects/amber/design-notes/patterns/pattern-matching-for-java`. Automatic garbage collection was included in Java in 1995; the functional language LISP had this already in 1958. Generics were added to Java 5 in 2004; the functional language SML had it since 1990. Higher-order functions were added to C# in 2007, to Java 8 in 2014; again LISP had them since 1958. Also Rust, a C-like programming language that has been developed since 2010 and is gaining quite some interest, borrows many ideas from functional programming from yesteryear.

A Scala program for generating pretty pictures of the Mandelbrot set.
(See https://en.wikipedia.org/wiki/Mandelbrot_set or
https://www.youtube.com/watch?v=aSg2Db3jF_4):



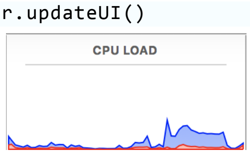
sequential version:



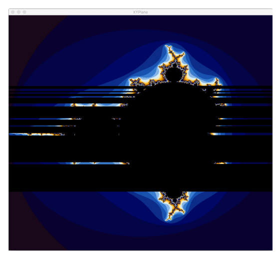
```
for (y <- (0 until H)) {
  for (x <- (0 until W)) {

    val c = start +
      (x * d_x + y * d_y * i)
    val iters = iterations(c, max)
    val colour =
      if (iters == max) black
      else colours(iters % 16)

    pixel(x, y, colour)
  }
  viewer.updateUI()
}
```



parallel version on 4 cores:



```
for (y <- (0 until H).par) {
  for (x <- (0 until W).par) {

    val c = start +
      (x * d_x + y * d_y * i)
    val iters = iterations(c, max)
    val colour =
      if (iters == max) black
      else colours(iters % 16)

    pixel(x, y, colour)
  }
  viewer.updateUI()
}
```

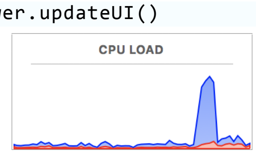


Figure 2: The code of the two "main" loops in my version of the mandelbrot program. The parallel version differs only in `.par` being added to the "ranges" of the x and y coordinates. As can be seen from the CPU loads, in the sequential version there is a lower peak for an extended period, while in the parallel version there is a short sharp burst for essentially the same workload…meaning you get more work done in a shorter amount of time. This easy *parallelisation* only works reliably with immutable programs.

lowing article about FP (functional programming) — you might have to disable your browser cookies though if you want to read it for free. And spoiler alert: This is tongue-in-cheek `;o)`

> https://archive.ph/vrofC

Relevant xkcd entries about functional programming are XXX.

The Very Basics

Let us get back to Scala: One advantage of Scala over Java is that it includes an interpreter (a REPL, or Read-Eval-Print-Loop) with which you can run and test small code snippets without the need of a compiler. This helps a lot with interactively developing programs. It is my preferred way of writing small Scala programs. Once you installed `scala`, you can start the interpreter by typing on the command line:

```
$ scala
Welcome to Scala 3.7.3 (21.0.8, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala>
```

The precise response may vary depending on the version and platform where you installed `scala`. Make sure however that `scala` uses version 3—you can find the version number in the welcome message. Also note that at the first time `scala` runs, it might download various components, for example the Scala compiler, Scala runtimes etc. Once `scala` is up and running, you can type at the prompt expressions like `2 + 3` Ret and the output will be

```
scala> 2 + 3
val res0: Int = 5
```

The answer means that he result of the addition is of type `Int` and the actual result is 5; `res0` is a name that Scala gives automatically to the result. You can reuse this name later on, for example

```
scala> res0 + 4
val res1: Int = 9
```

Another classic example you can try out is

```
scala> println("hello world")
hello world
```

Note that in this case there is no result! The reason is that `println` does not actually produce a result (there is no `resX` and no type), rather it is a function that causes the *side-effect* of printing out a string. Once you are more familiar with the functional programming-style, you will know what the difference is between a function that returns a result, like addition, and a function that causes a side-effect, like `println`. We shall come back to this point later, but if you are curious now, the latter kind of functions always has `Unit` as return type. It is just not printed by Scala.

You can try more examples with the `scala` REPL, but feel free to first guess what the result is (not all answers by Scala are obvious):

```
scala> 2 + 2
scala> 1 / 2
scala> 1.0 / 2
scala> 1 / 2.0
scala> 1 / 0
scala> 1.0 / 0.0
scala> true == false
scala> true && false
scala> 1 > 1.0
scala> "12345".length
scala> List(1,2,1).size
scala> Set(1,2,1).size
scala> List(1) == List(1)
scala> Array(1) == Array(1)
scala> Array(1).sameElements(Array(1))
scala> 0.1 + 0.2 == 0.3
```

If you think Scala's answer in the last line is braindamaged, try the same in your own favourite language. Also observe carefully what Scala responds in the following three instances involving the constant `1`—can you explain the differences?

```
scala> 1
scala> 1L
scala> 1F
```

Please take the Scala REPL seriously: If you want to take advantage of my reference implementation for the assignments, you will need to be able to "play around" with it!

Standalone Scala Apps

If you want to write a standalone app in Scala, you can implement a function `hello` and annotate the tag `@main`. For example write

```
@main
def Hello() = println("hello world")
```

save it in a file, say `hello-world.scala`, and then use `scala` (which compiles the scala file and runs it):

```
$ scala hello-world.scala
hello world
```

Like Java, Scala targets the JVM and consequently Scala programs can also be executed by the bog-standard Java Runtime. This can be done as follows:

```
$ scala --power package --assembly hello-world.scala
$ java -jar Hello.jar
hello world
```

Values

> Do not use **var** in your code for PEP! This declares a mutable variable. Only use **val**! This is for *immutable* values.

In the lectures I will try to avoid as much as possible the term *variables* familiar from other programming languages. The reason is that Scala has *values*, which can be seen as abbreviations of potentially larger expressions. The keyword for defining values is **val**. For example

```
scala> val x = 42
val x: Int = 42

scala> val y = 3 + 4
val y: Int = 7

scala> val z = x / y
val z: Int = 6
```

As can be seen, we first define `x` and `y` with admittedly some silly expressions, and then reuse these values in the definition of `z`. All easy, right? Why the kerfuffle about values? Well, values are *immutable*. You cannot change their value after you defined them. If you try to reassign `z` above, Scala will yell at you:

```
scala> z = 9
-- [E052] Type Error: -----------------------------------
1 |z = 9
  |^^^^^
  |Reassignment to val z
  | ...
1 error found
```

So it would be a bit absurd to call values as variables...you cannot change them; they cannot vary. You might think you can reassign them like

```
scala> val x = 42
scala> val z = x / 7
scala> val x = 70
scala> println(z)
```

but try to guess what Scala will print out for `z`? Will it be `6` or `10`? A final word about values: Try to stick to the convention that names of values should be lower case, like `x`, `y`, `foo41` and so on. Upper-case names you should reserve for what is called *constructors*. And forgive me when I call values as variables…it is just something that has been in imprinted into my developer-DNA during my early years and is difficult to get rid of. `;o)`

Function Definitions

We do functional programming! So defining functions will be our main occupation. As an example, a function named `f` taking a single argument of type `Int` can be defined in Scala as follows:

```
def f(x: Int) : String = ...YOUR CODE...
```

This function returns the value resulting from evaluating the expression what your code is. Since we declared `String` after the colon, the result of this function will be of type `String`. It is a good habit to always include this information about the return type, while it is only strictly necessary to give this type in recursive functions (later more on that). Simple examples of Scala functions are:

```
def incr(x: Int) : Int = x + 1
def double(x: Int) : Int = x + x
def square(x: Int) : Int = x * x
```

The general scheme for functions is

```
def fname(arg1: ty1, arg2: ty2,..., argn: tyn): rty = {
  ...BODY_OF_FUNCTION...
}
```

where each argument, `arg1`, `arg2` and so on, requires its type and the result type of the function, `rty`, should also be given. If the body of the function is more complex, then it can be enclosed in braces, like above. If it is just a simple expression, like `x + 1`, you can omit the braces. Very often functions are recursive (that is call themselves), like the venerable factorial function:

```
def fact(n: Int) : Int =
  if (n == 0) 1 else n * fact(n - 1)
```

In this case we have to give the return type `Int`. But as said, it is a good habit to give the return type for all functions. Note we could also have written this with braces as

```
def fact(n: Int) : Int = {
  if (n == 0) 1
  else n * fact(n - 1)
}
```

but this seems a bit overkill for a small function like `fact`. Notice that I did not use a `then`-keyword in the `if`-statements and that I enclosed the condition inside parentheses, like `(n == 0)`. Your eyes might hurt to not see an `then` with an `if`, but this has been long established syntax for `if`-statements. Scala, to my knowledge, was pretty unique in that for nearly 20 years of its existence…until Scala 3 came along. While people like me have perfectly adapted to the sight of `if`s without `then`s, it seems the developers of Scala caved in to the special eyesight of Gen-Python people (I am sure that is not you) and now allow to write the same function also as

```
def fact(n: Int) : Int = {
  if n == 0 then 1
  else n * fact(n - 1)
}
```

The main difference between both versions is that if you want to drop the `then`, then you need to enclose the boolean expression within parentheses. I accept the second version might look a bit more familiar to beginners of Scala, if they come from other languages, like Python, Java or C++. But that we also had to get rid in Scala 3 of the familiar `{}`-parentheses is completely beyond me. So in Scala 3 the braces are optional and the `fact`-function can even be written as

```
def fact(n: Int) : Int =
     if n == 0
     then 1
     else n * fact(n - 1)
```

on the condition that indents now become *meaningful* (as in Python).🤮 All versions are now permitted in Scala 3. While you are free to use any syntax version you want in PEP, or even mix them, I will **not** show you any of my code in the newfangled Pythonesque meaningful-indent-syntax. When necessary, I will always use braces to indicate the beginning and end of a code block, and I have not yet completely got used to the `if`s with `then`s. Please forgive me for being still inconsistent with this[5]

However, no matter which syntax style you adopt for `if`s, never write an `if` without an `else`-branch! That has something to do with functional programming and you will see its purpose later on. Coming back to function definitions: While `def` is the main mechanism for defining functions, there are a few other ways for doing this. We will see some of them in the next sections.

Before we go on, let me explain one tricky point in function definitions, especially in larger definitions. What does a Scala function return as result? Scala has a `return` keyword, but it is used for something different than in Java (and C/C++). Therefore please make sure no `return` slips into your Scala code.

So in the absence of `return`, what value does a Scala function actually produce? A rule-of-thumb is whatever is in the last line of the function is the value

[5]Scala adopted some very fine features of Python, for example string interpolations, but that we had to completely cave in to the demands of Gen-Python is a bridge too far for my completely insignificant opinion. I always assumed escaping Python's dependency hell is every software developers life goal—apparently not. 🤯

that will be returned. Consider the following example:[6]

```
def average(xs: List[Int]) : Int = {
  val s = xs.sum
  val n = xs.length
  s / n
}
```

In this example the expression `s / n` is in the last line of the function—so this will be the result the function calculates. The two lines before just calculate intermediate values. This principle of the "last-line" comes in handy when you need to print out values, for example, for debugging purposes. Suppose you want rewrite the average function as

```
def average(xs: List[Int]) : Int = {
  val s = xs.sum
  val n = xs.length
  val h = xs.head
  println(s"Input $xs with first element $h")
  s / n
}
```

Here the function still only returns the expression `s / n` in the last line. The `println` before just prints out some information about the input of this function, but does not contribute to the result of the function. Similarly, the value `h` is used in the `println` but does not contribute to what integer is returned by the function.

A caveat is that the idea with the "last line" is only a rough rule-of-thumb. A better rule might be: the last expression that is evaluated in the function. Consider the following version of `average`:

```
def average(xs: List[Int]) : Int = {
  if (xs.length == 0) 0
  else xs.sum / xs.length
}
```

What does this function return? Well there are two possibilities: either the result of `xs.sum / xs.length` in the last line provided the list `xs` is nonempty, **or** if the list is empty, then it will return `0` from the **`if`**-branch (which is technically not the last line, but the last expression evaluated by the function in the empty-case).

Summing up, do not use **`return`** in your Scala code! A function returns what is evaluated by the function as the last expression. There is always only one such last expression. Previous expressions might calculate intermediate values, but they are not returned. If your function is supposed to return multiple things, then one way in Scala is to use tuples. For example returning the

[6] We could have written this function in just one line, but for the sake of argument let's keep the two intermediate values.

minimum, average and maximum can be achieved by

```
def avr_minmax(xs: List[Int]) : (Int, Int, Int) = {
  if (xs.length == 0) (0, 0, 0)
  else (xs.min, xs.sum / xs.length, xs.max)
}
```

which still satisfies the rule-of-thumb: The result of the function is the last expression that is run inside the function.

> Do not use `return` in your code to indicate what a function produces as a result! It has a different meaning in Scala than in Java. It can change the meaning of your program, and you should never use it.

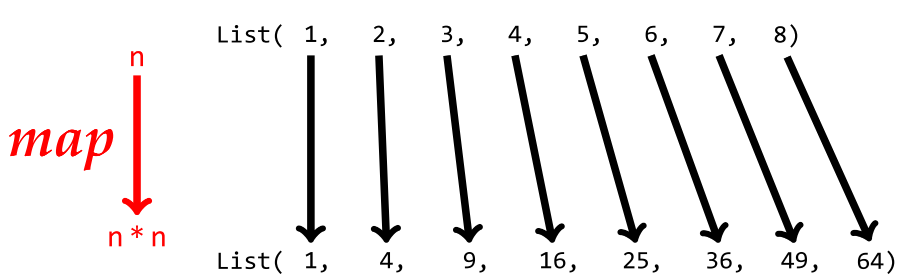
Loops, or Better the Absence Thereof

Coming from Java or C/C++, you might be surprised that Scala does not really have loops. It has instead, what is in functional programming called, *maps*. To illustrate how they work, let us assume you have a list of numbers from 1 to 8 and want to build the list of corresponding squares. The list of numbers from 1 to 8 can be constructed in Scala as follows:

```
scala> (1 to 8).toList
val res1: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8)
```

Like in modern versions of Java, the `1 to 8` generates a `Range`, which is then transformed into a list by the `.toList`. Generating from this list the list of squares in an imperative programming language such as C++, you would assume the list is given as a kind of array. You would then iterate, or loop, an index over this array and replace each entry in the array by its square. Right? In Scala, and in other functional programming languages, you use maps to achieve the same.

A map essentially takes a function that describes how each element is transformed (in this example the function is $n \rightarrow n * n$) and a list over which this function should work. Pictorially you can think of the idea behind maps as follows:



On top is the "input" list we want to transform; on the left is the "map" function for how to transform each element in the input list (the square function in this case); at the bottom is the result list of the map. This means that a map generates a *new* list, unlike a for-loop in Java or C/C++ which would most likely just update the existing list/array.

Now there are two ways for expressing such maps in Scala. The first way is called a *for-comprehension*. The keywords are **for** and **yield**. Squaring the numbers from 1 to 8 with a for-comprehension would look as follows:

```
scala> for (n <- (1 to 8).toList) yield n * n
val res2: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64)
```

This for-comprehension states that from the list of numbers we draw some elements. We use the name `n` to range over these elements (whereby the name is arbitrary; we could use something more descriptive if we wanted to). Using `n` we compute the result of `n * n` after the **yield**. This way of writing a map resembles a bit the for-loops from imperative languages, even though the ideas behind for-loops and for-comprehensions are quite different. Also, this is a simple example—what comes after **yield** can be a complex expression enclosed in `{...}`. A more complicated example might be

```
scala> for (n <- (1 to 8).toList) yield {
         val i = n + 1
         val j = n - 1
         i * j + 1
       }
val res3: List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64)
```

Let us come back to the simple example of squaring a list of numbers from above. As you can see in the for-comprehensions, we specified the list where each `n` comes from, namely `(1 to 8).toList`, and how each element needs to be transformed, the expression after the **yield**. This can also be expressed in a second way in Scala by using directly the function `map` as follows:

```
scala> (1 to 8).toList.map(n => n * n)
val res3 = List(1, 4, 9, 16, 25, 36, 49, 64)
```

In this way, the expression `n => n * n` stands for the function that calculates the square (this is how the `n`s are transformed by the map). It might not be obvious, but the for-comprehensions above are just syntactic sugar: when compiling such code, Scala translates for-comprehensions into equivalent maps. This even works when for-comprehensions get more complicated (see below).

The very charming feature of Scala is that such maps or for-comprehensions can be written for any kind of data collection, such as lists, sets, vectors, options and so on. For example if we instead compute the remainders modulo 3 of this list, we can write

```
scala> (1 to 8).toList.map(n => n % 3)
val res4 = List(1, 2, 0, 1, 2, 0, 1, 2)
```

If we, however, transform the numbers 1 to 8 not into a list, but into a set, and then compute the remainders modulo 3 we obtain

```
scala> (1 to 8).toSet.map(n => n % 3)
val res5 = Set(2, 1, 0)
```

This[7] is the correct result for sets, as there are only three equivalence classes of integers modulo 3. Since maps and for-comprehensions are just syntactic variants of each other, the latter can also be written as

```
scala> for (n <- (1 to 8).toSet) yield n % 3
val res5 = Set(2, 1, 0)
```

For-comprehensions can also be nested and the selection of elements can be guarded (or filtered). For example if we want to pair up the numbers 1 to 4 with the letters a to c, we can write

```
scala> for (n <- (1 to 4).toList;
            m <- ('a' to 'c').toList) yield (n, m)
val res6 = List((1,a), (1,b), (1,c), (2,a), (2,b), (2,c),
                (3,a), (3,b), (3,c), (4,a), (4,b), (4,c))
```

In this example the for-comprehension ranges over two lists, and produces a list of pairs as output. Or, if we want to find all pairs of numbers between 1 and 3 where the sum is an even number, we can write

```
scala> for (n <- (1 to 3).toList;
            m <- (1 to 3).toList;
            if (n + m) % 2 == 0) yield (n, m)
val res7 = List((1,1), (1,3), (2,2), (3,1), (3,3))
```

The `if`-condition in this for-comprehension filters out all pairs where the sum is not even (therefore `(1, 2)`, `(2, 1)` and `(3, 2)` are not in the result because their sum is odd).

To summarise, maps (or for-comprehensions) transform one collection into another. For example a list of `Int`s into a list of squares, and so on. There is no need for for-loops in Scala. But please do not be tempted to write anything like

```
scala> val cs = ('a' to 'h').toList
scala> for (n <- (0 until cs.length).toList)
          yield cs(n).capitalize
val res8: List[Char] = List(A, B, C, D, E, F, G, H)
```

This is accepted Scala-code, but utterly bad style (it is more like Java). It can be written much clearer as:

```
scala> val cs = ('a' to 'h').toList
scala> for (c <- cs) yield c.capitalize
val res9: List[Char] = List(A, B, C, D, E, F, G, H)
```

[7] This returns actually `HashSet(1, 2, 3)`, but this is just an implementation detail of how sets are implemented in Scala.

Results and Side-Effects

While hopefully all this about maps looks reasonable, there is one complication: In the examples above we always wanted to transform one list into another list (e.g. list of squares), or one set into another set (set of numbers into set of remainders modulo 3). What happens if we just want to print out a list of integers? In these cases the for-comprehensions need to be modified. The reason is that `print`, you guessed it, does not produce any result, but only produces what is in the functional-programming-lingo called a *side-effect*…it prints something out on the screen. Printing out the list of numbers from 1 to 5 would look as follows

```
scala> for (n <- (1 to 5).toList) print(n)
12345
```

where you need to omit the keyword **`yield`**. You can also do more elaborate calculations before printing such as

```
scala> for (n <- (1 to 5).toList) {
  val square = n * n
  println(s"$n * $n = $square")
}
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
```

In this code I use a value assignment (`val square = ...`) and also what is called in Scala a *string interpolation*, written `s"..."`. The latter is for printing out formatted strings. It allows me to refer to the integer values `n` and `square` inside a string. This is very convenient for printing out "things".

The corresponding map construction for functions with side-effects is in Scala called `foreach`. So you could also write

```
scala> (1 to 5).toList.foreach(n => print(n))
12345
```

or even just

```
scala> (1 to 5).toList.foreach(print)
12345
```

If you want to find out more about maps and functions with side-effects, you can ponder about the response Scala gives if you replace `foreach` by `map` in the expression above. Scala will still allow `map` with side-effect functions, but then reacts with a slightly interesting result. If you understand the difference, you are pretty much on the road of becoming a master-functional programmer.

Aggregates

There is one more usage of for-loops in Java, C/C++ and the like: sometimes you want to *aggregate* something about a list, for example summing up all its elements. In this case you cannot use maps, because maps *transform* one data collection into another data collection. They cannot be used to generate a single integer representing an aggregate. So how is this kind of aggregation done in Scala? Let us suppose you want to sum up all elements from a list. You might be tempted to write something like

```
var cnt = 0
for (n <- (1 to 8).toList) {
  cnt += n
}
print(cnt)
```

and indeed this is accepted Scala code and produces the expected result, namely `36`, **BUT** this is imperative style and not permitted in PEP. If you submit this kind of code, you get 0 marks. The code uses a `var` and therefore violates the immutability property I ask for in your code. Sorry!

So how to do that same thing without using a `var`? Well there are several ways. One way is to define the following recursive `sum`-function:

```
def sum(xs: List[Int]) : Int =
  if (xs.isEmpty) 0 else xs.head + sum(xs.tail)
```

You can then call `sum((1 to 8).toList)` and obtain the same result without a mutable variable and without a for-loop. Obviously for simple things like sum, you could have written `xs.sum` in the first place. But not all aggregate functions are pre-defined and often you have to write your own recursive function for this.

Higher-Order Functions

Functions obviously play a central role in functional programming. Two simple examples are

```
def even(x: Int) : Boolean = x % 2 == 0
def odd(x: Int) : Boolean = x % 2 == 1
```

More interestingly, the concept of functions is really pushed to the limit in functional programming. Functions can take other functions as arguments and can return a function as a result. This is actually quite important for making code generic. Assume a list of 10 elements:

```
val lst = (1 to 10).toList
```

Say, we want to filter out all even numbers. For this we can use

```
scala> lst.filter(even)
List(2, 4, 6, 8, 10)
```

where `filter` expects a function as argument specifying which elements of the list should be kept and which should be left out. By allowing `filter` to take a function as argument, we can also easily filter out odd numbers as well.

```
scala> lst.filter(odd)
List(1, 3, 5, 7, 9)
```

Such function arguments are quite frequently used for "generic" functions. For example it is easy to count odd elements in a list or find the first even number in a list:

```
scala> lst.count(odd)
5
scala> lst.find(even)
Some(2)
```

Recall that the return type of `even` and `odd` are booleans. Such function are sometimes called predicates, because they determine what should be true for an element and what false, and then performing some operation according to this boolean. Such predicates are quite useful. Say you want to sort the `lst`-list in ascending and descending order. For this you can write

```
lst.sortWith(_ < _)
lst.sortWith(_ > _)
```

where `sortWith` expects a predicate as argument. The construction `_ < _` stands for a function that takes two arguments and returns true when the first one is smaller than the second. You can think of this as elegant shorthand notation for

```
def smaller(x: Int, y: Int) : Boolean = x < y
lst.sortWith(smaller)
```

Say you want to find in `lst` the first odd number greater than 2. For this you need to write a function that specifies exactly this condition. To do this you can use a slight variant of the shorthand notation above

```
scala> lst.find(n => odd(n) && n > 2)
Some(3)
```

Here `n => ...` specifies a function that takes `n` as argument and uses this argument in whatever comes after the double arrow. If you want to use this mechanism for looking for an element that is both even and odd, then of course you out of luck.

```
scala> lst.find(n => odd(n) && even(n))
None
```

While functions taking functions as arguments seems a rather useful feature, the utility of returning a function might not be so clear. I admit the following example is a bit contrived, but believe me sometims functions produce other functions in a very meaningful way. Say we want to generate functions according to strings, as in

```
def mkfn(s: String) : (Int => Boolean) =
  if (s == "even") even else odd
```

With this we can generate the required function for `filter` according to a string:

```
scala> lst.filter(mkfn("even"))
List(2, 4, 6, 8, 10)
scala> lst.filter(mkfn("foo"))
List(1, 3, 5, 7, 9)
```

As said, this is example is a bit contrived—I was not able to think of anything simple, but for example in the Compiler module next year I show a compilation functions that needs to generate functions as intermediate result. Anyway, notice the interesting type we had to annotate to `mkfn`. The types in Scala are described next.

Types

In most functional programming languages, types play an essential role. Scala is such a language. You have already seen built-in types, like `Int`, `Boolean`, `String` and `BigInt`, but also user-defined ones, like `Rexp` (see coursework). Unfortunately, types can be a thorny subject, especially in Scala. For example, why do we need to give the type to `toSet[Int]`, but not to `toList`? The reason is the power of Scala, which sometimes means it cannot infer all necessary typing information. At the beginning, while getting familiar with Scala, I recommend a "play-it-by-ear-approach" to types. Fully understanding type-systems, especially complicated ones like in Scala, can take a module on their own.[8]

In Scala, types are needed whenever you define an inductive datatype and also whenever you define functions (their arguments and their results need a type). Base types are types that do not take any (type)arguments, for example `Int` and `String`. Compound types take one or more arguments, which as seen earlier need to be given in angle-brackets, for example `List[Int]` or `Set[List[String]]` or `Map[Int, Int]`.

Scala provides a basic mechanism to check the type of a (closed) expression—closed means that all parts are already known to Scala. Then you can use the command `:type` and check in the REPL:

```
scala> :type (1, List(3), Set(4,5), "hello")
(Int, List[Int], Set[Int], String)
```

If Scala can calculate the type of the given expression, then it will print it. Unfortunately, this way of finding out a type is almost unusable: for 'things' where

[8]Still, such a study can be a rewarding training: If you are in the business of designing new programming languages, you will not be able to turn a blind eye to types. They essentially help programmers to avoid common programming errors and help with maintaining code.

the type is pretty obvious, it gives an answer; but for 'things' that are actually of interest (such as what is the type of a pre-defined function), it gives up with an error message.

There are a few special type-constructors that fall outside this pattern. One is for tuples, where the type is written with parentheses. For example

```
(Int, Int, String)
```

is for a triple (a tuple with three components—two integers and a string). Tuples are helpful if you want to define functions with multiple results, say the function returning the quotient and remainder of two numbers. For this you might define:

```
def quo_rem(m: Int, n: Int) : (Int, Int) =
  (m / n, m % n)
```

Since this function returns a pair of integers, its *return type* needs to be of type `(Int, Int)`. Incidentally, this is also the *input type* of this function. For this notice `quo_rem` takes *two* arguments, namely `m` and `n`, both of which are integers. They are "packaged" in a pair. Consequently the complete type of `quo_rem` is

```
(Int, Int) => (Int, Int)
```

This uses another special type-constructor, written as the arrow `=>`. This is sometimes also called *function arrow*. For example, the type `Int => String` is for a function that takes an integer as input argument and produces a string as result. A function of this type is for instance

```
def mk_string(n: Int) : String = n match {
  case 0 => "zero"
  case 1 => "one"
  case 2 => "two"
  case _ => "many"
}
```

It takes an integer as input argument and returns a string. The type of the function generated in `mkfn` above, is `Int => Boolean`.

Unfortunately, unlike other functional programming languages, there is in Scala no easy way to find out the types of existing functions, except by looking into the documentation

https://dotty.epfl.ch/api/index.html

The function arrow can also be iterated, as in `Int => String => Boolean`. This is the type for a function taking an integer as first argument and a string as second, and the result of the function is a boolean. Though silly, a function of this type would be

```
def chk_string(n: Int)(s: String) : Boolean =
  mk_string(n) == s
```

which checks whether the integer `n` corresponds to the name `s` given by the function `mk_string`. Notice the unusual way of specifying the arguments of this function: the arguments are given one after the other, instead of being in a pair (what would be the type of this function then?). This way of specifying the arguments can be useful, for example in situations like this

```
scala> List("one", "two", "three", "many").map(chk_string(2))
res4 = List(false, true, false, false)

scala> List("one", "two", "three", "many").map(chk_string(3))
res5 = List(false, false, false, true)
```

In each case we can give to `map` a specialised version of `chk_string`—once specialised to 2 and once to 3. This kind of "specialising" a function is called *partial application*—we have not yet given to this function all arguments it needs, but only some of them.

Coming back to the type `Int => String => Boolean`. The rule about such function types is that the right-most type specifies what the function returns (a boolean in this case). The types before that specify how many arguments the function expects and what their type is (in this case two arguments, one of type `Int` and another of type `String`). Given this rule, what kind of function has type `(Int => String) => Boolean`? Well, it returns a boolean. More interestingly, though, it only takes a single argument (because of the parentheses). The single argument happens to be another function (taking an integer as input and returning a string). Remember that `mk_string` is just such a function. So how can we use it? For this define the somewhat silly function `apply_3`:

```
def apply_3(f: Int => String): Bool = f(3) == "many"

scala> apply_3(mk_string)
res6 = true
```

You might ask: Apart from silly functions like above, what is the point of having functions as input arguments to other functions? Well, in all functional programming languages, including Scala, it is really essential to allow functions as input argument. Above you have already seen `map` and `foreach` which need this feature. Consider the functions `print` and `println`, which both print out strings, but the latter adds a line break. You can call `foreach` with either of them and thus changing how, for example, five numbers are printed.

```
scala> (1 to 5).toList.foreach(print)
12345
scala> (1 to 5).toList.foreach(println)
1
2
3
4
5
```

This is actually one of the main design principles in functional programming. You have generic functions like `map` and `foreach` that can traverse data containers, like lists or sets. They then take a function to specify what should be done with each element during the traversal. This requires that the generic traversal functions can cope with any kind of function (not just functions that, for example, take as input an integer and produce a string like above). This means we cannot fix the type of the generic traversal functions, but have to keep them *polymorphic*.[9]

There is one more type constructor that is rather special. It is called `Unit`. Recall that `Boolean` has two values, namely **`true`** and **`false`**. This can be used, for example, to test something and decide whether the test succeeds or not. In contrast the type `Unit` has only a single value, written `()`. This seems like a completely useless type and return value for a function, but is actually quite useful. It indicates when the function does not return any result. The purpose of these functions is to cause something being written on the screen or written into a file, for example. This is what is called they cause a *side-effect*, for example new content displayed on the screen or some new data in a file. Scala uses the `Unit` type to indicate that a function does not have a result, but potentially causes a side-effect. Typical examples are the printing functions, like `print`.

More Info

There is much more to Scala than I can possibly describe in this short document and teach in the lectures. Fortunately there are a number of free books about Scala and of course lots of help online. For example

- https://www.youtube.com/user/ShadowofCatron
- http://docs.scala-lang.org/tutorials
- https://www.scala-exercises.org
- https://twitter.github.io/scala_school

There is also an online course at Coursera on Functional Programming Principles in Scala by Martin Odersky, the main developer of the Scala language. And a document that explains Scala for Java programmers

- http://docs.scala-lang.org/tutorials/scala-for-java-programmers.html

While I am quite enthusiastic about Scala, I am also happy to admit that it has more than its fair share of faults. For example, whilst implicits are great, they can also be a source of great headaches, for example consider the code:

```
scala>  List (1, 2, 3) contains "your mom"
res1: Boolean = false
```

[9] Another interesting topic about types, but we omit it here for the sake of brevity.

Scala Syntax for Java Developers

Scala compiles to the JVM, like the Java language. Because of this, it can re-use many libraries. Here are a few hints how some Java code tranlsates to Scala code:

Variable declaration:

```java
Drink coke = getCoke();
```
Java

```scala
val coke : Drink = getCoke()
```
Scala

or even

```scala
val coke = getCoke()
```
Scala

Unit means void:

```java
public void output(String s) {
  System.out.println(s);
}
```
Java

```scala
def output(s: String): Unit = println(s)
```
Scala

Compound types, say the type for list of Strings:

```java
List<String>
```
Java

```scala
List[String]
```
Scala

String interpolations

```java
System.out.println("Hello, "+ first + " "+ last + "!");
```
Java

```scala
println(s"Hello, $first $last!")
```
Scala

Java provides some syntactic sugar when constructing anonymous functions:

```java
list.foreach(item -> System.out.println("* " + item));
```
Java

In Scala, we use the **=>** symbol for the same:

```scala
list.foreach(item => println(s"* $item"))
```
Scala

Rather than returning `false`, this code should throw a typing-error. There are also many limitations Scala inherited from the JVM that can be really annoying. For example a fixed stack size. One can work around this particular limitation, but why does one have to? More such 'puzzles' can be found at

`http://scalapuzzlers.com` and `http://latkin.org/blog/2017/05/02/when-the-scala-compiler-doesnt-help/`

Even if Scala has been a success in several high-profile companies, there is also a company (Yammer) that first used Scala in their production code, but then moved away from it. Allegedly they did not like the steep learning curve of Scala and also that new versions of Scala often introduced incompatibilities in old code. Also the Java language is lately developing at lightening speed (in comparison to the past) taking on many features of Scala and other languages, and it seems it even introduces new features on its own. So there is seemingly even more incentive to stick with the old stuff you know. Still, the goal of this part of PEP is to bend your mind about what programming is…namely functional programming. I promise you, this will be useful no matter with which programming language you will work.

Scala is deep: After many years, I still continue to learn new technique for writing more elegant code. Scala 3 seems to iron out a number of snags from Scala 2, but why on earth are they introducing Python-esque indentation and why on earth are they re-introducing the `then`-keyword in Scala 3, when I just about got comfortable without it?

Conclusion

I hope you liked the short journey through the Scala language—but remember we like you to take on board the functional programming point of view, rather than just learning another language: Immutable functions are easier to trust, because they the same output on the same input. For the same reason they are easier to test and debug. There is an interesting blog article about Scala by a convert:

`https://www.skedulo.com/tech-blog/technology-scala-programming/`

He makes pretty much the same arguments about functional programming and immutability (one section is teasingly called *"Where Did all the Bugs Go?"*). If you happen to moan about all the idiotic features of Scala (3), well, I guess this is part of the package according to this quote:

> *There are only two kinds of languages: the ones people complain about and the ones nobody uses.*
>
> *—Bjarne Stroustrup (the inventor of C++)*