# Example: Finding Fixed Points

Principles of Functional Programming

# Finding a fixed point of a function

A number `x` is called a *fixed point* of a function `f` if

```
f(x) = x
```

For some functions `f` we can locate the fixed points by starting with an initial estimate and then by applying `f` in a repetitive way.

```
x, f(x), f(f(x)), f(f(f(x))), ...
```

until the value does not vary anymore (or the change is sufficiently small).

# Programmatic Solution

This leads to the following function for finding a fixed point:

```
val tolerance = 0.0001

def isCloseEnough(x: Double, y: Double) =
  abs((x - y) / x) < tolerance

def fixedPoint(f: Double => Double)(firstGuess: Double): Double =
  def iterate(guess: Double): Double =
    val next = f(guess)
    if isCloseEnough(guess, next) then next
    else iterate(next)
  iterate(firstGuess)
```

# Return to Square Roots

Here is a *specification* of the `sqrt` function:

> `sqrt(x)` = the number `y` such that `y * y = x`.

Or, by dividing both sides of the equation with `y`:

> `sqrt(x)` = the number `y` such that `y = x / y`.

Consequently, `sqrt(x)` is a fixed point of the function `(y => x / y)`.

# First Attempt

This suggests to calculate `sqrt(x)` by iteration towards a fixed point:

```
def sqrt(x: Double) =
  fixedPoint(y => x / y)(1.0)
```

Unfortunately, this does not converge.

Let's add a `println` instruction to the function `fixedPoint` so we can follow the current value of `guess`:

# First Attempt (2)

```
def fixedPoint(f: Double => Double)(firstGuess: Double) =

  def iterate(guess: Double): Double =
    val next = f(guess)
    println(next)
    if isCloseEnough(guess, next) then next
    else iterate(next)

  iterate(firstGuess)
```

`sqrt(2)` then produces:

```
2.0
1.0
2.0
1.0
...
```

# Average Damping

One way to control such oscillations is to prevent the estimation from varying too much. This is done by *averaging* successive values of the original sequence:

```
def sqrt(x: Double) = fixedPoint(y => (y + x / y) / 2)(1.0)
```

This produces

```
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.4142135623746899
```

In fact, if we expand the fixed point function `fixedPoint` we find a similar square root function to what we developed last week.

# Functions as Return Values

The previous examples have shown that the expressive power of a language is greatly increased if we can pass function arguments.

The following example shows that functions that return functions can also be very useful.

Consider again iteration towards a fixed point.

We begin by observing that $\sqrt{x}$ is a fixed point of the function `y => x / y`.

Then, the iteration converges by averaging successive values.

This technique of *stabilizing by averaging* is general enough to merit being abstracted into its own function.

```
def averageDamp(f: Double => Double)(x: Double): Double =
  (x + f(x)) / 2
```

# Exercise:

Write a square root function using `fixedPoint` and `averageDamp`.

# Final Formulation of Square Root

```
def sqrt(x: Double) = fixedPoint (averageDamp (y => x/y)) (1.0)
```

This expresses the elements of the algorithm as clearly as possible.

# Summary

We saw last week that functions are essential abstractions because they allow us to introduce general methods to perform computations as explicit and named elements in our programming language.

This week, we've seen that these abstractions can be combined with higher-order functions to create new abstractions.

As a programmer, one must look for opportunities to abstract and reuse.

The highest level of abstraction is not always the best, but it is important to know the techniques of abstraction, so as to use them when appropriate.