Software Lab 4 *6.01 – Fall 2011*

Signal Class

Goals:

The overall goal is to implement a family of Python classes which demonstrate PCAP principles with signals. We will:

1. Explore creation and representation of simple signals, represented as subclasses of a **Signal class**.
2. Build complex signals from primitive ones, using a universal set of elementary methods for transforming signals.

1 Setup

Using your own laptop

- Be sure you have the 6.01 software libraries installed.
- Download and unzip `swLab04.zip` into a convenient folder (e.g., `~/Desktop/6.01/swLab04`).

To use the plotting features of today's software, you must invoke `idle` using the `-n` switch:
On a lab laptop, MacOS, or Linux: type the following command in a terminal window:
`> idle -n`
On Windows: download and unzip the `idle-n.bat` file (see the 6.01 Software page). Then, start idle by running `idle-n.bat`.

2 Introduction

The software lab for this week explores the world of signals, represented in Python as a class named `Signal`. We seek to understand this representation, and appreciate how arbitrary signals may be constructed from primitive signals, using a universal set of elementary operations for transforming signals.

2.1 The Signal class

Objective: Explore the python `Signal` representation of abstract signals, by creating simple signals and plotting them.

Resources:

- **Section 5.2** of the course notes.
- On the 6.01 OCW Scholar site, go to the Software and Tools section, and click on 6.01 Software Documentation. Navigate to the documentation for the **`sig` module**.
- `swLabs04WorkA.py`: template for working with the `Signal` class.

The `Signal` class provides methods for *representing* a signal, for graphically *depicting* a signal, and for *combining* signals to produce new signals.

Any signal can be thought of as being an infinite sequence of sample values. Thus, in the `Signal` class, a signal is represented as a function which can be called with any time index; it returns the value of the signal at that index. Specifically, `Signal` assumes that every instance has a method called `sample`, which takes time index, `n`, and returns the value of the signal at time `n`. Thus, subclasses of `Signal` may be constructed, each with the `sample` method implemented to produce different signals, as appropriate.

The simplest signal we could imagine having is one with a constant value. Here is its class definition:

```
class ConstantSignal(Signal):
    def __init__(self, c):
        self.c = c
    def sample(self, n):
        return self.c
```

The `UnitSample` signal has value 1 at time 0 and value 0 at all other times:

```
class UnitSampleSignal(Signal):
    def sample(self, n):
        if n == 0:
            return 1
        else:
            return 0
```

Somewhat more interesting is a sinusoidal signal; we implement this with a `CosineSignal` class, whose constructor takes a frequency `omega` and a phase `phase`:

```
class CosineSignal(Signal):
    def __init__(self, omega = 1, phase = 0):
        self.omega = omega
        self.phase = phase
    def sample(self, n):
        return math.cos(self.omega * n + self.phase)
```

A method `plot(self, start=0, end=100, newWindow='...', color='blue')` is also defined for all signals. It plots the signal from `start` to `end`, which should be integers. If `newWindow` is specified, it is a string usef for the title of the new window for this plot; if `color` is specified, it as a string specifying the color of the new curve to be added to the plot. The parameters `start`, `end`, `newWindow` and `color` all have reasonable default arguments, so you can just specify the ones you need.

Plotting will *only* work from `idle` if you start it with the `-n` flag. Currently, the first signal plotted in a new window sets its size; plot the signal with the largest range first.

2.2 Explore simple Signals

Step 1.

- Open the file `swLabs04WorkA.py` in Idle. It imports our implementation of the module `sig`, which defines all of the classes and methods described in this handout. So, you can use those classes and methods in calls that you make in this file. For instance, to make a unit sample signal, you can do:

  ```
  s = UnitSampleSignal()
  ```

- Add a Python command in that file that will make a plot of the unit sample signal. If `s` is a signal, then
  `s.plot(-5, 5)`
  will plot that signal in a new window, with samples from time steps -5 to 5.
- Choose Run Module in Idle (under the Run menu in the window with the `swLab04WorkA.py` file). You should see the plot. Be sure it makes sense to you.
- Now make a cosine signal and plot it, in the same way. Try different values of `omega` and `phase`.

3 PCAP with Signals

Objective:

- Combine and transform primitive signals to produce more complex ones
- Implement step signal, and scale, sum, and delay classes for transforming signals

Resources:

- `swLabs04WorkA.py`: template for working with the `Signal` class.
- `swLab04SignalDefinitions.py`: template for new class definitions for combining signals
- `swLab04WorkB.py`: template for code to test your new class definitions

Five subclasses of `Signal` are described below, which provide a primitive "step" signal, and classes for summing two signals, scaling a signal, and delaying a signal. You will start by exploring how these work, using our implementation of the subclasses. Then we will ask you to implement your own version of them.

- `StepSignal()`: has value 0 at any time index less than 0 and value 1 otherwise.
- `SummedSignal(s1, s2)`: takes two signals, `s1` and `s2`, at initialization time and creates a new signal that is the sum of those signals. Note that this class should be *lazy*: when the signal is created, no addition should happen; values only need to be added when a sample of the summed signal is requested.
- `ScaledSignal(s, c)`: takes a signal `s` and a constant `c` at initialization time and creates a new signal whose sample values are the sample values of the original signal multiplied by `c`.
- `R(s)`: takes a signal `s` at initialization time and creates a new `Signal` whose samples are delayed by one time step; that is, the value at time `n` of the new signal should be the value at time `n` $-$ 1 of the old signal.
- `Rn(s, n)`: takes a signal `s` at initialization time and creates a new `Signal` whose samples are delayed by `n` steps.

Polynomials in R

We have built up an **algebra of signals**, which is very general. We can add, scale, and delay signals. One way to express combinations of these operations on a signal is to describe them in terms of a polynomial in $\mathcal{R}$. So, we will implement a procedure to construct a new signal by operating on it with a combination of sums, scales, and delays described by a polynomial in $\mathcal{R}$.

So, if X is a signal, then we can describe a delayed version of X as $\mathcal{R}$X, and a version of X that has been delayed by two time steps as $\mathcal{R}\mathcal{R}$X. That same signal, scaled by 7, would be $7\mathcal{R}\mathcal{R}$X or $7\mathcal{R}^2$X. If we added $7\mathcal{R}^2$X to X, then we'd have $7\mathcal{R}^2\text{X} + \text{X}$, which we can think of as the signal X transformed by a polynomial with coefficients `[7, 0, 1]`. So, we could define a procedure

- `polyR(s, p)`: takes an instance of the `Signal` class (or one of its subclasses), `s`, and an instance of the `Polynomial` class, `p`, and returns a new signal that is `s` transformed by a polynomial in $\mathcal{R}$.

3.1 From primitive to complex Signals

First, we will use an existing implementation of the signal constructors to build complex signals from primitive ones, in order for you to develop an intuition about how they work.

Detailed guidance:

Step 2. Do tutor problem **Wk.4.1.1**, using the following steps:

- Edit the file `swLab04WorkA.py` in Idle to construct new instances of signals by making combinations (e.g., sums, scalings, and delays) of instances of the basic `UnitSampleSignal` and `StepSignal` classes, as specified in the tutor problem. **You can use any of the classes or procedures in the `sig` module. You should not write any new Python classes for this problem!!**
- Plot them to be sure they're right.
- Remember that the `polyR` procedure takes as a second argument an instance of the `Polynomial` class. You can create new instances of this class via `poly.Polynomial(c)` where `c` is a list of coefficients.

Wk.4.1.1 Do this tutor problem on constructing signals.

3.2 Combining Signals – under the hood

Now, we will implement the classes we just used to make complex signals out of simple ones. Our implementations should have the same behavior as what you experienced in the section above.

Detailed guidance:

Step 3. The remaining problems ask you to implement each of the new subclasses of `Signal`, as described above. Each one only needs to supply an appropriate `sample` method, and possibly an `__init__` method. We have provided you with a file, `swLab04SignalDefinitions.py`, that you can use to debug your implementations before entering them in the Tutor. You should enter your new class definitions at the end of `swLab04SignalDefinitions.py`. You can test them by putting your test cases in `swLab04WorkB.py`, which imports `swLab04SignalDefinitions.py`, and using Run Module on `swLab04WorkB.py` in Idle.

You can only use the methods and classes that are defined in `swLab04SignalDefinitions.py`; you will extend that file to be a complete implementation of the `sig` module.

It is very important that, for testing, you use `swLab04WorkB.py` which imports your class definitions, rather than `swLab04WorkA.py`, which imports our class definitions.

Check Yourself 1. Whenever you define a new signal class, make an instance or two and plot them to be sure you're getting something reasonable.

Wk.4.1 Do tutor problems Wk.4.1.2 through Wk.4.1.4

Check Yourself 2. We have defined `usamp` to be an instance of the `UnitSample` class. What does the signal `polyR(usamp, poly.Polynomial([3, 5, -1, 0, 0, 3, -2])` look like?
Do you see how you could use `polyR` to construct any signal (with finitely many non-zero values) out of the unit sample?

Step 4. Implement the **procedure** `polyR(s, p)`, which takes an instance of the `Signal` class (or one of its subclasses), `s`, and an instance of the `Polynomial` class, `p`, and returns a new signal that is `s` transformed by a polynomial in $\mathcal{R}$. So, for example, if `p` is an instance of the `Polynomial` class, with `p.coeffs = [3, 2, 10]`, then the result of `polyR(s, p)` is a signal

$$3\mathcal{R}\mathcal{R}s + 2\mathcal{R}s + 10s$$

In your implementation, you can use any of the previously implemented signal constructors, including `Rn`. Note that you don't need to define a class to do this: just writing a single procedure will

suffice. Depending on how you choose to write your code, you may find it helpful to remember that any list, such as `test`, can be reversed by calling `test.reverse()`

Wk.4.1.5 Do tutor problem **Wk.4.1.5**, on the `polyR` procedure.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011