Research Shell

Maciej Trawka

December 26, 2022

Contents

Chapter 1

Overview

The Research Shell is in fact a Python3 shell wrapped by `PtPython` with some useful modules, classes and methods included. This document covers those items assuming, that a reader is familiar with Python syntax.

1.1 Architecture of Research Shell

Look at Figure 1.1. It shows Research Shell wrappers from the top to the Python3 core. User calls a Bash script, which prepares and executes a command containing Python3 call, module PtPython loading, and then importing all useful libraries included in module `aio`. So to run the Research Shell you need to call:

```
research_shell [python_file_name_to_execute]
```

If no argument, then the Research Shell appears and is ready to execute Python commands. If a script file is specified as an argument, then its content is executed after importing all modules and the shell closes.

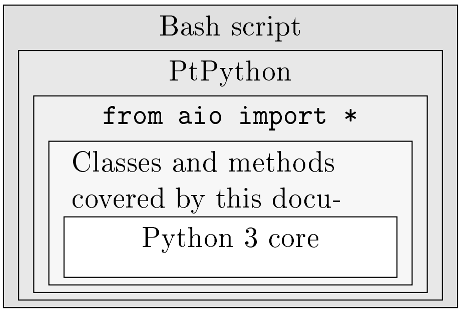

Figure 1.1: Research Shellarchitecture.

There is also a special mode of Research Shell, called **Testcase Mode**. It makes easy to execute a complete testcases. The testcase is a directory having a regular structure:

```
testcase_name
 |__ data ........................ automatically added to the searching path
 |__ results ..................... Created automatically by Research Shell
 |__ driver.py ................... Main script - your Testcase code
```

By running the command:

```
research_shell_drun
```

the Research Shell runs in the Testcase mode. In such case it checks if the `driver.py` file exists. If so, then it removes and recreates the `results` directory and goes there (so `results` is now the Current Directory). Now, a content of `dirver.py` is executed. In `results` directory a `transcript.txt` is created. To print something to the screen and also to the transcript file, you need to call the `print(*args)` method of class `Aio`, i.e.:

```
# This text will be printed to the screen only:
print("Text␣on␣the␣screen␣only")

# This also appears in the transcript file:
Aio.print("Text␣on␣the␣screen␣and␣in␣the␣transcript")
```

Chapter 2

Class Polynomial

`Polynomial` is an object intended to analyze polynomials over GF(2). An object of type `Polynomial` holds polynomial coefficients (as a list of positive integers) and a list of signs of those coefficients. Of course in case of GF(2) coefficient $x_i = -x_i$. However, negative coefficients make sense in case of some types of LFSRs, as `Polynomial` objects are used to create other objects, of type of `Lfsr`.

Below you can see an example of how to create a `Polynomial` object representing the polynomial $x^{16} + x^5 + x^2 + x^0$:

```
p1 = Polynomial ( [16, 5, 2, 0] )
p2 = Polynomial ( 0b10000000000100101 )
p2 = Polynomial ( 0x10025 )
```

`Polynomial` class includes also a couple of static methods, especially useful to search for primitive polynomials and other ones discussed in the next part of this chapter.

2.1 Polynomial object methods

`str(<Polynomial>)`

`Polynomial` objects are convertible to strings.

```
p1 = Polynomial ( [16, 5, 2, 0] )
print(p1)
# >>> [16, 5, 2, 0]
```

`hash(<Polynomial>)`

`Polynomial` objects are hashable. Can be used as a dictionary keys.:

```
p1 = Polynomial ( [16, 5, 2, 0] )
d = {}
d[p1] = "p1␣value"
```

`<Polynomial>.getCoefficients()`

Returns a reference to coefficients list of the `Polynomial` object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
coeffs1 = p1.getCoefficients()
print(coeffs1)
# >>> [16, 5, 2, 0]
coeffs.remove(0)
print(coeffs1)
# >>> [16, 5, 2]
print(p1)
# >>> [16, 5, 2]
```

`<Polynomial>.getCoefficientsCount()`

Returns count of the `Polynomial` object coefficients.

```
p1 = Polynomial ( [16, 5, 2, 0] )
coeffscount1 = p1.getCoefficientsCount()
print(coeffscount1)
# >>> 4
```

`<Polynomial>.getDegree()`

Returns degree of the `Polynomial` object.

```
p1 = Polynomial ( [16, 5, 2, 0] )
deg1 = p1.getDegree()
print(deg1)
# >>> 16
```

2.2 Static Polynomial methods

Index