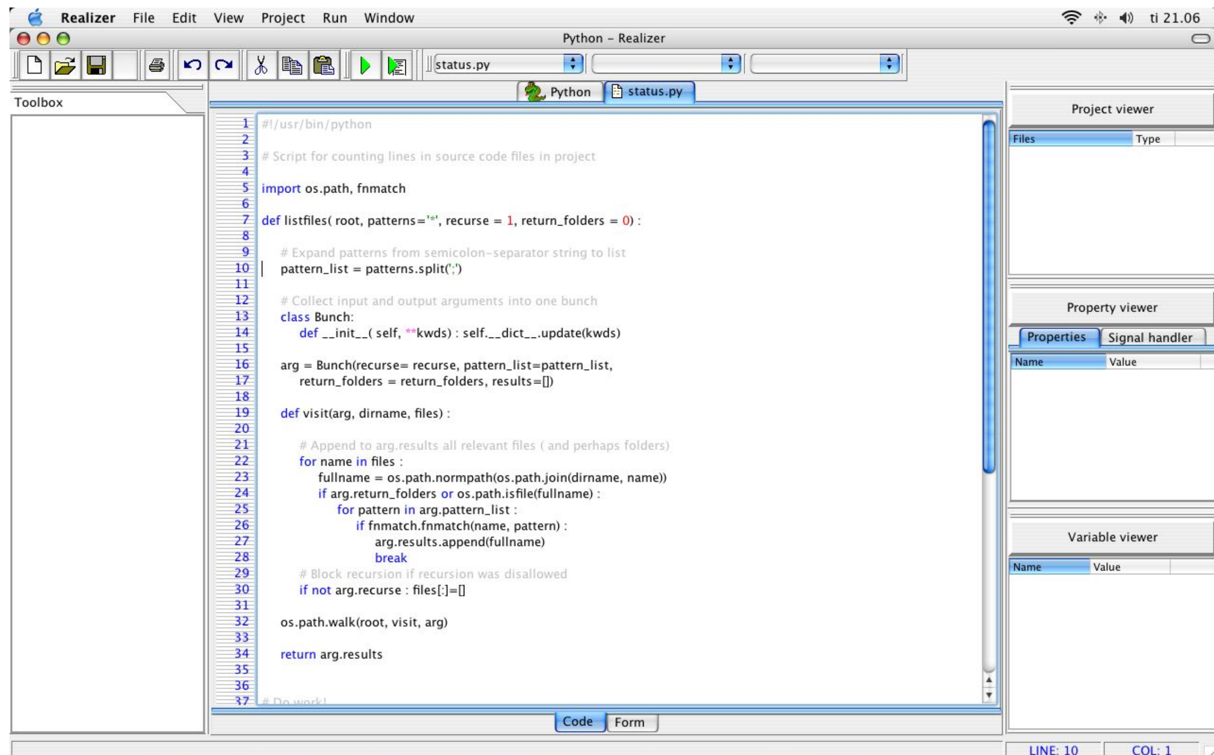
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NORA - Norwegian Open Research Archives

Python - Realizer



Rapid application development with Python rewritten in C++ and Qt.

http://www.python-realizer.net

Master thesis at

Simula research laboratory and University of Oslo, Norway.

2004 - 2005

Credits

There are several people I would like to thank for letting me finish my master thesis and thereby finishing my master degree in computer science.

First of all, I would like to thank my parents for given me the environment for studying and teaching me the value of working hard to get what I want, even when the odds are against one.

I especially want to thank the unemployment agency of Norway and all the persons who supported me on my quest to transfer from a job, my health could not support in the long run to better opportunities in computer science.

My employer AS Oslo Sporveier has been very helpful in the transfer process, and made it possible for me to take five year of my life and use it for pursuing a higher education. It gave me complete attention to my school duties without going broke.

And even though my health, cut my career short, it has been a very good place to work for over fifteen years, both as a part time worker in the beginning while studying to become an electrical engineer and later as a full time position.

I would also take this opportunity to thank all my good friends at the University of Oslo, both teachers and good student friends. They all made me feel at home at campus from day one. I will miss them deeply when we all move on.

I have been so lucky to get a great Python guru as my master-thesis advisor. He is always positive to my suggesting and even when he is opposed to my ideas, he still is positive in his response. He made me forget my ideas gently with a smile. His name is Kent Andre Mardal by the way.

He still has not convinced me that math is easy, and I still don't understand his math drawings on the office walls and his fondness of the word "the".

I must not forget to thank Simula research laboratory for giving me a great working environment for finishing my master thesis. It is a great place to work, with great people.

Trolltech has for the last 10 years or so, created the best class library for C++ programming ever, and I am very thankful for they given me such a great tool to work with. It really makes C++ a great development language.

And last I will thank all the smart people behind creating the original Python system, which has been and still are a great inspiration for my own system.

Table of contents

Credits 2
Table of contents 3
Summary 4
Introduction 6
Development system 12
  General application functionality 13
    Toolbars 14
    Menus 15
    Utility windows 16
  The syntax coloring subsystem ... 18
  The editor component 18
  The interpreter component 18
Python parser in C++ 20
  Lexical analyzing 22
    Reserved keywords 23
    Operators 23
    Delimiters 23
    Special meaning tokens 24
    Literals used in Python 24
    Implementation details 24
    Data structure TOKENID 26
    Data structure FULLTOKEN 26
    Collecting text to analyze 27
  Abstract syntax tree 28
    Definition of node structure 28
    Definition of node type 29
    Statement nodes 30
    Expression nodes 30
    Container nodes 31
    Literal nodes 31
    Example node generation 31
  Parser 32
    Start from user input 32
    Start from file or buffer 32
    Compound statements 33
    Simple statements 35
    Expressions 37
    Data structures in Parser 39
The Python type system 41
  Abstract data type 42
  Boolean data type 42
  Buffer data type 43
  Cell data type 44
  Class data type 44
  Code data type 44
  Complex data type 45
  Description data type 46
  Dictionary data type 46
  Enumeration data type 47
  File data type 47
  Float data type 48
  Frame data type 48
  Function data type 49
  Generator data type 49
  Integer data type 49
  Iterator data type 50
  List data type 50
  Long data type 51
  Method data type 52
  Module data type 52
  None data type 53
  Object base type 53
  Range data type 54
  Set data type 54
  Slice data type 54
  String data type 55
  Struct data type 56
  Tuple data type 56
  Get arguments system 57
  Build data objects system 59
The error system 60
Compiler system 61
  Marshal system 61
Virtual executing machine 63
  Operand codes 63
  Implementation details 64
Extensions modules in C++ 66
The road ahead 67
Appendix A 69
  Full grammar of Python 2.4 69
Appendix B 72
  Building of the application and install Qt. 72
Literature reference 73
Colophon 74

Summary

Python – Realizer is basically planned to become the original Python done in C++ and Qt[1] class library. This is to make it a brand new implemented Python interpreter with a new rapid application development application extending it, for a complete Python system.

It is supposed to become the Visual Basic of Python, in a matter of speak with most ideas taken from that environment. At least what it was in the earlier edition, before everything got transfer to Visual Studio.

It should not include any original C code from the C based standard Python system, but is based heavily on the standard C Python language and its library functions. Most of the library modules written in the Python language will later be possible to compile and use directly in the new C++ based system.

I am trying to be as compatible as possible with Python version 2.4 in language and library modules, except for when that is unwise in regards to making it a C++ system.

It will for one thing be based on Unicode character strings using Qt library's QString class, and therefore does not have a standard string module and a special Unicode module. That would not make sense in the context of Qt framework.

Further we use C++ operator overloading heavily for access to Realizer built-ins data types, such that the built-in data type's can be used as standard C++ classes outside of Python programs. And since we are rewriting the standard C based Python system in C++, everything is a class and we use C++ data types like *bool* instead of larger *int* data type like C must do.

The application I make for encapsulating the new Python interpreter is of course portable and will become a total development system, with code editors for writing Python scripts.

Form editors will later be used for visually creating of dialog windows and program the main window, without bothering with all the small details.

To summarize the project, we will strive to include all the great functions found in other environment for software development, which we found useful for our purpose.

The original C based Python system was originally and still is, written by a large group of people, who spend several years to get the system to where it is now. Not to mention it will take a lot of work to get a full development application ready for Python.

It is of course not possible for me to finish making Python – Realizer a complete system during just a year work on my master-thesis.

The purpose of my master-thesis is however to create the foundation for this system, such that other master students can work to extend it, or that this can become an open source project.

[1] Qt is a class library for GUI and more, made by Trolltech in Norway. http://www.trolltech.com

I concentrate in the beginning on the work on the application, with an interpreter that at least can parse the Python code given from the user or from a file, and check that for syntactical errors found in conflict with valid Python grammars. Further I will start implementing a code editor with syntax coloring of code as a helper to write good Python code.

Also I will start and work as much as I can with the very important library containing Python built-in data type subsystem, that in fact do a lot of Pythons work.

I also start to work on a second library, which will contain the important virtual execute machine for running Python code in a cross platform way.

The rest of the system will be described for other to implement at a later time, or at least what must be done to complete the interpreter and the library modules.

And you will always found the updated source code for this project at homepage:

http://www.python-realizer.net/

Make sure you get the latest code by checking the archive files date before downloading. It is named in the following template:

Realizer_MMM_DD_2005.tar.gz

Where "MMM" is month and "DD" is the day.

The one really important goal of my project is to make a platform independent development system with Python at its center. I program mainly on the Apple's Macintosh computers, but will make sure it compiles and run under Linux and to some extend even on a Windows based computer.

It also includes the application with code editors and more. The easiest way to make sure this is possible, is to choose a class library that support cross platform development out of the box, without having to worry about the difference between platforms.

I found that in Trolltech's Qt class library, and it is free for none commercial use. It also has great classes predefined for what I need to implement in the Python interpreter and all the supporting libraries. It really makes the code shorter, when the functionality you want already is implemented in ready to use classes.

And last, C++ is a very powerful language to write my system in, and it creates native executable code for each platform I want to support without the slowdown of executing byte code in virtual machines like Java and C# mostly do. With Qt library, C++ finally can compete with the standard libraries included with those other languages.

The first try to make PR is found in the file *old_PythonRealizer.tar*.gz and will also contain the source files, not yet converted to the current version.

Introduction

How we program computer systems, has changed greatly through time, from the earliest computer giants build around the end of the Second World War and up to these days' powerful personal desktop computers.

I have myself, have the pleasure of follow the gradually development of computers from early 80's with its small compact home computers and all the way to present all dominating personal computers.

In the computers childhood, it was basically only possible to program a computer by switching switches on and off, to program instructions and possibly data into a machine.

A famous 30-year-old computer, which was based on this principle, is the 1975 based *Altair 8800*[2] computer, which contributed greatly to Microsoft existents today.

I remember spending Christmas evening in 1984, by manually type in the small programs as simple games found in home computer magazines, in the form of numbers in a Basic program. The numbers was a primitive way to program machine code without an assembler.

Basic-interpreters built-in in these small home computers was not powerful enough to run even small games, fast enough for gamers to be accepted.

And the only way to use machine code in your program in those days was to manually enter them as boring lists of numbers into a simple Basic program, which saved them into memory as machine codes. There was not enough memory to run an Assembler in those early days of home computing.

In those early years of my computer experience, the computer of choice was the famous *Commodore Vic-20* with 5 kilobytes of memory, where 3.5 kilobytes where free to use for your own programs. There was no storage system available to me at this time. I could not afford a cassette player for storing programs, and I had to manually write the program into the computer every time I wanted to run a program from paper.

At the end of the day, you turned the computer off, and the program was lost forever.

If you wanted to play another game, you turned the power off and then on again, and started to write the next program into memory from paper line by line.

This was not an ideal way to program computer systems to put it mildly, and it made sure the program was simple and small to fit into the computer memory. With the storage system like paper based card and tapes, it took forever to get a program loaded into memory. It limited the usefulness of computers, and it was important to find a better way to program computers with much better tools.

The next step in programming the computer in those days was to use an Assembler program to translate from the textual representation of machine

[2] Per A. Holst book in Norwegian : "Datateknologiens utvikling", By Tapir akademisk forlag 2001 Page 501 – 502.

code to the binary numbers the computer knew how to execute.

The assembler translates, as you know simple verbal instructions to their binary representation.

It was an improvement over the old methods to get machine code into computers, but still it was not possible to move programs from one computer system to another system with a different processor. Those days, there where lots of computer systems with different processors, making it very hard to write portable code.

It also toke a lot of time to write even simple programs, with several lines of assembler code, just to make simple program constructs. Even with the later macro assemblers with a little support for abstraction, it was very hard to develop complex programs in a reasonably time frame.

It would be very clear, that we needed a computer language to program in, that was more like a human written language. One needs to translate from a reasonably understandable language to the computer machine code for executing.

A program to do this task is called a compiler, and today they exist in large numbers for different source language and destination machine code, but where rare earlier.

The biggest computer manufacture in those days was IBM. They produced a large number of different computer systems, and often with different instruction sets.

It was important to find a better way to move program between those systems without having to write a new version for each machine type.

A group of computer scientist working at IBM's research center in 1954 developed a high-level computer language called *Fortran*[3] for use in the technical and scientific areas.

It was short for "Formula translation" which described the language purpose in short terms. It is a very simple language compared to modern computer languages, and it is basically for math purpose, and it does that so well that even today, it is widely in use.

A standard for *Fortran* was finished as late as in 1995.

In the earliest day of computers, it was often used for administrative computer work. Fortran was not especially good at these kinds of tasks. So another language was designed for this purpose.

It was the language called Cobol[4]. The American department of defense developed it. Its name is short for "Common business oriented language".

This language is without a doubt, one of the most used computer language even to this day, and still a lot of programs in daily use are written in this old language.

Both these early computer language is not of the all purpose language type, but rather special adapted to their intended task and nothing much else.

One just choose the language best suited for the task one needed to do,

[3] Per A. Holst – Page 361 - 373
[4] Per A. Holst – Page 373 - 376

and not like today, based on other preferences like the personal taste or what is available from ones project leader or what ever.
As time went by, the need for more general programming language grew. One serious reason for this is that one didn't want to learn new languages all the time to solve different tasks.

History is full of more or less popular computer languages. Only a few survived into the present day, most are dead by now or only of historical interest.

I will only mention two important languages from the period starting at the end of 60's and to the end of 70's.

The first is the language *Basic*[5], which was invented at Dartmouth – college, in its first edition in 1964.

Its main goal, was to become an easy to learn computer language, which especially students and later home computer users, would find useful when programming their computer systems in the 70s and the 80s.

In the beginning, the Basic language was almost always an interpreter. Often built into the computers read only memory, when sold as home computers, and made an abstraction against the computers machine instructions.

The program was interpreted by this command interpreter, and made it possible to write mostly portable programs without having to bother with the underlined architecture used in each computer. There where a lot of different dialects of Basic and not all where compatible with each other.

Later, Basic language got compilers, which translated Basic programs into the real executable machine code. That made them a lot faster to execute and tailor made for each platform. An important example of this is the Visual Basic from Microsoft, which is still very popular on desktop computers, and is a very important inspiration for my project.

Even though the language was easy to learn and use, it was not very well suited for the more complex software development. Later edition of Basic is however "made" object oriented, and thereby a little more suited for the more complex software development. It also lets you make an application in a reasonable short time, especially when you are under a strict deadline.

The early edition of Basic often forced the programmer to write some function in machine code to speed up the execution of a program to a tolerated level.

But that made them both less portable and less easy to understand by reading the source code.

The other language I want to describe from those early years of programming, that still is heavily used today, is the very important language called C. The same people who invented the operating system Unix designed it in the early 70's.

This language is often called a system language, since it is used to program operating systems, compilers and other important programs. This is a high level language, but can often be mistaken for being closer to an assembler, which often is built-in in its compiler.

[5] Per A. Holst – Page 383 - 388

The two language described are both called procedural based language. It uses procedures to code the functionality it can reuse during the executing of programs. It has no native built-in encapsulating of data or functionality outside its use of procedures.

This makes it not very suited for very complex programs, and one often get name collisions in large programs between variables used in different part of the program having the same name.

A new and better way to encapsulate the data and the functionality in a program was needed, and thanks to the two Norwegian computer scientists, we now have what we call "Object oriented language", which almost every current modern computer language is designed to be.

The first language to use this principle was the Norwegian developed language called *Simula*[6].

This language was developed by the Norwegian computer center in Oslo, by Ole-Johan Dahl and Kirsten Nygård, with the first edition finished in 1965. It created the foundation for all modern object oriented languages.

Even though this language never got widely used, it had a very important influence on the languages like C++, C#, Java and others.

These languages are widely used today for all application and system development around the world, often with extensively class libraries as the standard part of the language.

This makes it possible to reuse old codes in new application, without having to invent the wheel every time you need one. The time and effort it takes to make a complex program, is drastically reduced, and even is the reason several complex application can be made at all.

Still, sometimes these languages are not what one need. When time factor is very limited and one need a program like now and only have one programmer or only have a little need for this program in the long run.

Quick and dirty programs to solve small task fast with limited recourses like people or machines. To solve this task, there have been developed several small languages called script languages.

Simple version of these languages, are shell-based systems like *Bash*[7] and all the other shells found today.

These languages are well suited for the small task in system administration of a computer with no fuss, and don't need to be translated before executing on a computer. They are interpreted by the shell system directly on a computer.

Still, there are tasks that need to be done quick and easy, but is too complicated for shell systems. Maybe the program should run on several different computer systems or maybe it is to be run in a web context.

A large group of scripting languages has been developed in the last 10 to 15 years to solve such tasks.

Examples are Tcl, Ruby, Perl and my personal favorite Python.

[6] Per A. Holst. Page 396 - 398

[7] Born again shell.

The programming language Python, or if you want the scripting language, was originally developed by a Dutch computer scientist by the name Guido Van Rossum. He still is the main man behind the Python system.

Python started its development around Christmas 1990, when Rossum needed a project to take him through the holydays. He stated developing Python on his Apple Macintosh computer, based on ideas he had about efficient and simple way to design and implement a computer language.

It was important to him, to design a system that easily could help building small to complex programs in a portable way, with easy to understand syntax. Only a few lines of code should accomplish a lot more than usual functionality in the other languages.

Most important of all, is that variables should not need to be defined before use, and that they could change dynamically during the executing of programs.

In addition to this dynamically type system, he wanted a large and rich library of functions in reusable modules. It should be object oriented like all the modern languages, but still easy and fast to use for the scripting purposes.

Later it got several GUI subsystems to help make programs with the windows style, and not only text based programs. One of the most important library for GUI in Python is borrowed from the language Tcl, and is called Tkinter

All the programs written in Python are interpreted by the command interpreter written in C for the executing of the byte code on the virtual executing engine.

You don't need to modify the source code written in Python to execute them on different platforms.

While Python system is great and that there are several development environment system for it, there are no one close to what the Visual basic provides.

Also a lot of great C++ class libraries exist, with GUI and other functionality that can be interesting to use in Python programs without having to use utilities like the Swig[8].

It can also be interesting to use Python's built in data types directly in C++ programs like them where standard C++ classes.

If we rewrite the Python system in C++, we can easily use C++ classes directly in Python, and if we make Python type system a class library, it makes it possible to let C++ programs use the Python types directly.

So as my master thesis project, I have decided to start on a very ambiguous plan to develop a complete Python system in C++ and using the Qt library as GUI for both the system and for the development application.

It will have its own Python interpreter written entirely from scratch in C++ with dynamically loadable libraries for built-ins data types and a virtual

[8] SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. More details at http:/www.swig.org

machine for the executing of byte code.

Around this interpreter I will start to design a "rapid application development" application for all your Python development needs.

My goal is to build the foundation for this system, and make it available for others to complete or turn it into an open source project.

One can always dream!

Development system

Python – Realizer are designed to be based on one integrated application for the development of the code, the visual elements and the interpreting of Python code entered in the interpreter window of the application or from the files in form of a compiled byte code file or a source code file written in Python.

The Python code is to be executed by the virtual byte code machine, which is planned to be a dynamical loadable library written in C++ for use by the application and maybe later by other programs.

All the built-in data types is to be made available for both the interpreter and C++ programs who want to use this functionality, by another dynamically loadable library.

The built in modules may be implemented as a third library at a later time. But all the modules written in Python will be usable directly without any conversion with a few exceptions described in a later chapter.

I start by describing what I would like to include in the main development application, and what functionality it should provide to the programmer. I will then describe in details each sub system needed to complete the system.

The application is a program compiled as Realizer, and should be started by the executing of this program.

The main application is divided into visual parts like menus, toolbars, utilities windows and a tabulated presentation of the interpreter window and the code / form editor windows for each Python script that is open for editing.

The application is designed to be custom configured to suite your needs for an efficient working environment. This means that each user can decide which utility windows, if any, to be present in the work environment or where they should stay for easy access during development.

There will always be an interpreter window available in the application at all times, and it will be the first window in the tabulator.

Editor windows will open for each of the open Python source code, and each window will be tabulated into one window for code editing and one for form editing, to be used for the visual design if needed.

Functionality wanted, but not yet started to be implemented, is an integrated debugger with variable viewer for running programs and the possibility of single stepping a program.

There will be a need for a project manager. This will be a subsystem, which control each projects Python scripts and user interface description files.

An easy way to add class, methods and other element of Python script by selecting the details in the dialog boxes, instead of writing it all manually each time one need a new class declared.

There will definitive be a need for an easy to use and navigate, help or document viewer for the online help.

The code editor should remember all the indents for easy alignment of code blocks while you program, and each time you access a class or modules name in the editor, you should get a popup list with all members available in that class.

When you write a method or function name and is not sure about the syntax for its arguments, you should automatically get syntax described to you as small yellow utility popup notes.

I am also thinking about a PIM[9] / PSM[10] modeling system for Python with SQL support. With this, one can design classes and database tables in a Platform in depended models described in *UML*[11] and SQL for automatically translation to the Platform depended models in the form of Python classes.

Let us describe the system component that I have started to implement and is present in the application now.

General application functionality

The Realizer application is at present time a simple Python code editor, with a parser, capable of syntax checking scripts or user input in the interpreter window.

Both the editor component and the interpreter window have a color syntax scheme for easy reading.

The code editor will later have a lot more feature, given it a chance to become a full feature editor with all the bells and whistle you want or need to manage your Python code.

For now, it is a simple code editor capable of loading, editing and saving Python scripts with a simple line number scheme and coloring of keywords and more.
In earlier attempts to write a Qt based application, I made the mistake to sink into the details of Qt without really planning ahead.

In my first attempt to write an interpreter window for my parser, I started to create a custom widget from the ground up with handling of the text writing, the scrolling of the text in the window and the color syntax handling.

This was a serious mistake. Writing custom widget is fine, but the way to do it, is to take a widget with some of the functions you want, and extend it yourself by sub classing it.

Then you get all the boring handling of the text almost for free, and you only need to override some methods as needed. I wasted over 3 month trying to get my own widget to work reliable and without flickering during the use.

My biggest problem was without a doubt, flickering during the operation of the widget. And it was not easy to get the scrolling to operate correct either.

It didn't help, that I lost all the source code during a transfer between machines. Maybe it was for the best, since I started to program from scratch with a much better approach to a solution.

It became clear to me after some thinking. That it must be a much

[9] PIM – Platform independent models.
[10] PSM – Platform depended models.
[11] Universal modeling language.

easier and more acceptable way to make the application without being bog down by the details.

Qt is an extendable class framework library, and it is designed to be sub classed for easily add functionality without having to reinvent the wheels every time it is needed.

The secret is to find a standard widget with the closest match to what one need to implement. I therefore choose to use *QTextEdit*[12] as base class for two of my most important components.

I will describe each components implementation later in this chapter, but first explain the applications user interface in details.

Toolbars

In my attempt to show you all, what the complete development system, may be to look at when it is more complete, I have implemented some functionality into my application.

There is at present time, a total of tree separate toolbars for easy access to the functionality in the application, in supplement to the menu system.

The first toolbar present in the application, and maybe a very important one for most users, is the standard toolbar. This is a grouping of mainly file-oriented functionality.

Not all of the buttons are in use at the moment, and it is missing context based viewing of the active elements at the present time. Later one will only see the active components, and all the other should be grayed out.

The ones in use already are the following:

We have first a button for creating a new editor document in its own window.

Then there is a button for opening pre existing Python source files into their own editor window, by given the user a dialog window for entering name and path to the file which one want to open for editing. Then there is a button for saving the current active editor document to a file with already given name and path.

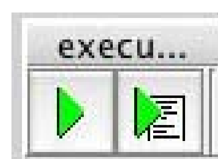
The next button is not yet in use, and is missing a good icon, but it will be for saving all the unsaved editor documents in one easy step.

The print subsystem is not implemented, but its button is showing only for demonstration purposes at this time.

The next two buttons are in use, and is for the undo / redo functionality in the editor windows in active Python scripts.

Then we have the standard buttons for cut, copy and paste functionality through the built-in clipboard. It is working between the edit documents and within it self.

The next toolbar, which is operational, though a bit thin in the functionality at this time, is the execute toolbar.



[12] This Widget is for basic text editing in Qt.

It has just two buttons as members now, but these two buttons starts executing or at least parsing of the source code written in Python.

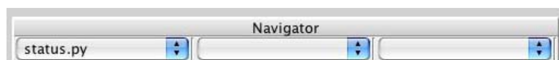
The first one, executes directly from the active editor window, and takes its buffer as input for the Parser.

The next one is for executing of Python scripts from files and the user gets a file dialog to choose which file to load into the Parser buffer.

Later we will add functionality for debugging of scripts, and running the scripts one instruction at the time or just to set a break point in the source code.

To ease the navigation, both between the edit documents and within a document for methods and classes, we have provided a navigator toolbar, and a dynamic windows menu.

First the navigator toolbar provides easy selection of scripts files opened in an editor window and later all its classes and of course methods of those classes.



At the moment only the selection of the active script file is implemented, and those files are also available in the windows menu.

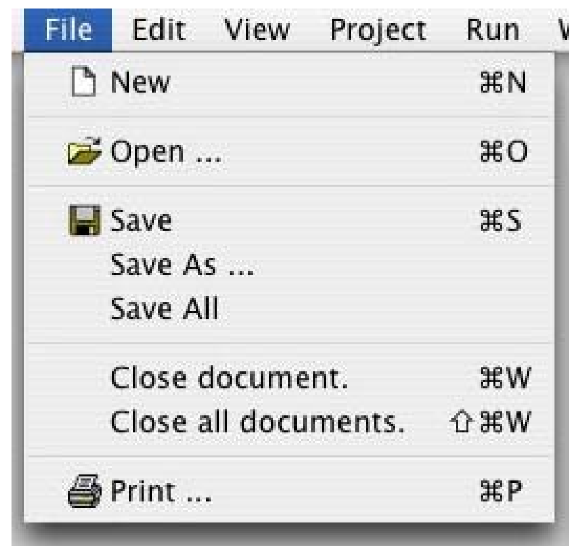
In the windows menu, you can also select easy access to the interpreter window, if you need that in a hurry.

You can of course also use the tab selecting widget on top of all the document windows and of the interpreter window.

Menus

The Realizer application has only limited menus implemented at this time, but I will explain the ones in use at present time, but all of the applications functionality will be available from the menu system in the final version.

We have most of the file menus functions available at the standard toolbar, but it also contains a few menu items not duplicated on toolbars.



First of all, we have more save possibilities in the menu than on the toolbar. We can save an open document to a file with a different name and possible new path.

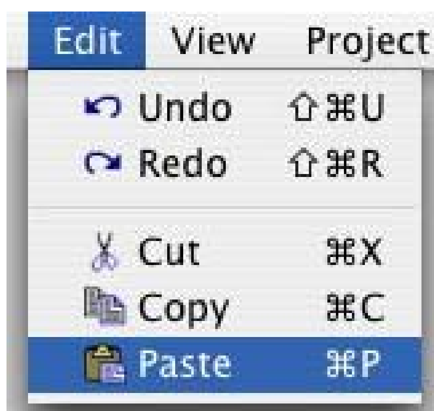
We have menu selections for closing either a single active editor document, or all opened editor documents. Later we will add the functionality for checking save status, before closing the documents.

Depending on which platform the application is running on, we might have exit the application on this menu bar, but on Macintosh computers, it will be at the application menu.

We need to add checking of all the opened and unsaved documents, before terminating the application, so we don't loose data when the application is terminated.

The next menu has also several items shared with the standard toolbar.

This is the edit menu, with the following functionality available at this time:



We have selections for the undo or redo of actions in the editor windows. This means that one can undo an action performed on a Python editor document, or redo it if you change your mind.

And we have copy, cut and paste functionality related to the same Python documents. It can also be used for copy and paste between the interpreter window and the editor documents. You can also use cut functionality in the interpreter window.

Next we have a view menu bar, which provide the user of the application with the options to view or hide part of the user interface. You can choose which toolbar to view or hide, and the same on utilities windows, which can dock to most of the edges of the application window.

Then we have a project menu, which is not yet implemented. But it will contain a subsystem for collecting the script files and the user interface description files into a logical unit.

Then we have the run menu, which only have two selections at this time. It is the same as the execute toolbar.

Then we have a dynamic menu bar called windows. It will have a selection for activating the interpreter window, or select one of the active editor windows with opened Python documents. It has a lot in common to the navigator toolbar. It is somewhat in operation already, in the form that we can select active windows, and it updates list with all possible editor windows available to be selected.

At last we have the help menu on some platforms, but not on Macintosh since that platform use an application menu bar for the same function.

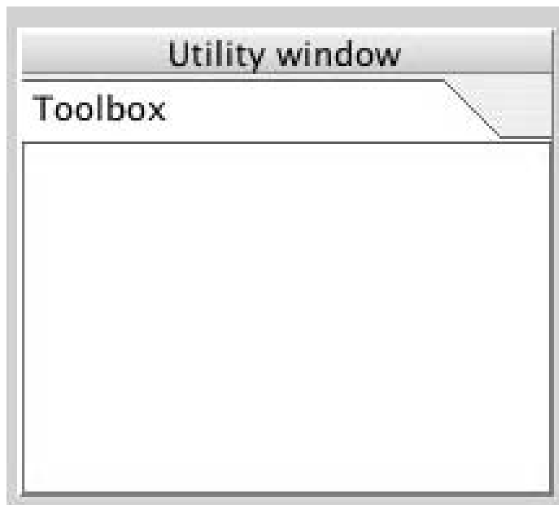
It only has a dialog window for information about the application, and a dialog for information about the version of Qt in use.

Utility windows

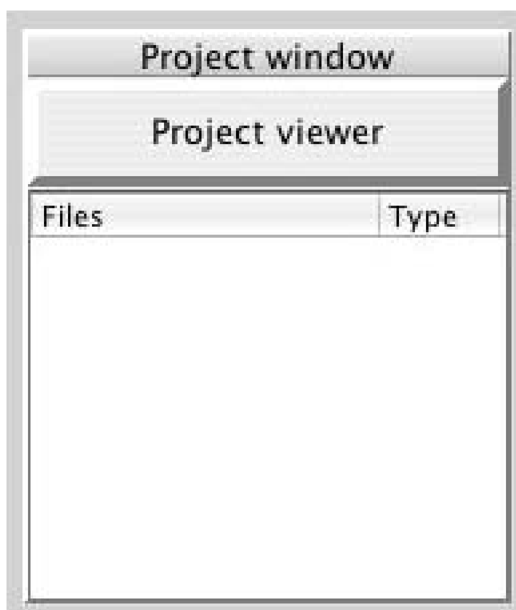
The Realizer application already has a lot of utility windows for added functionality, and will probably get more, later as we add functionality to the application.

None of them do any useful work at this time, but are present for prototyping of the user interface and to show you how I will the application to be presented in later versions.

Currently there are four different utility windows, and those are:

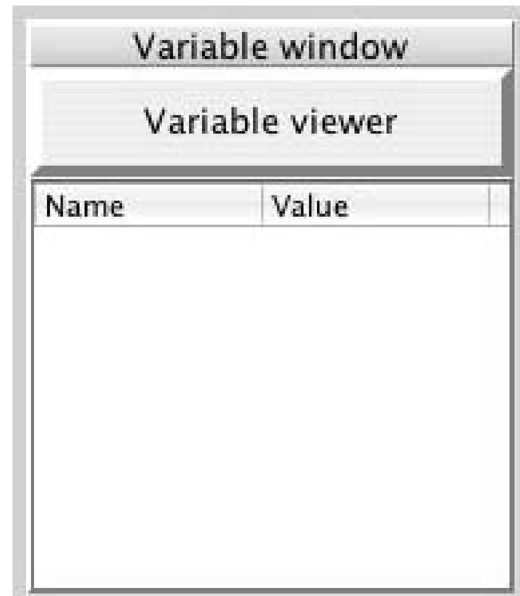


The utility-window, which will among other things, contains the selection window for widget in the user form, dialog templates and other useful things in the process of form designing.



Then we have a project viewer, which later will show all files and type of those files in a list for easy access, and to give the user an overview of the whole project at all time.

This will be connected to the project menu later.



This will be a kind of debug window, where a user can see what currently used variables contains during the executing of scripts either in step by step mode or during the standard executing of scripts.

At last we have the property window, which will be highly coupled to the form designer. It will let the user select properties for each widget placed on a form, and to control signal handlers for Qt's signal / slot mechanism.

The syntax coloring subsystem

Both components described later in this chapter, uses the same syntax coloring system for the highlighting of Python source code.

This coloring system makes all reserved keywords stand out in blue from the general text in black.

Numbers will be colored in red for easy spotting in the text. Valid operators in Python will be colored in light red. All comments will be in light gray and strings will be in light green.

Together all this coloring of the Python source code, makes it much easier to read a source code and navigated to where you need to alter or add code to your scripts.

All this functionality is implemented by sub classing the Qt's *QSyntaxHighlighter* class, which is then connected to the *QTextEdit* sub classed class used for the editor component. It will be described shortly.

It is only one simple method, we need to overload and write to get all this functionality almost for free.

It is the "highlightParagraph-method" which gets a single line and the previous lines status as input parameters, and returns this lines status when finished coloring the current line.

It then uses a simple lexical analyzer, which I wrote, to collect the tokens to be colorized by calling the "setFormat" method for the actual coloring.

I use a simple *QMap* dictionary for all the reserved keywords, as an easy and fast lookup table to decide if we need a blue colored text or not.

It is that easy to use the pre designed classes and subclass them for just a small change, to get the result needed.

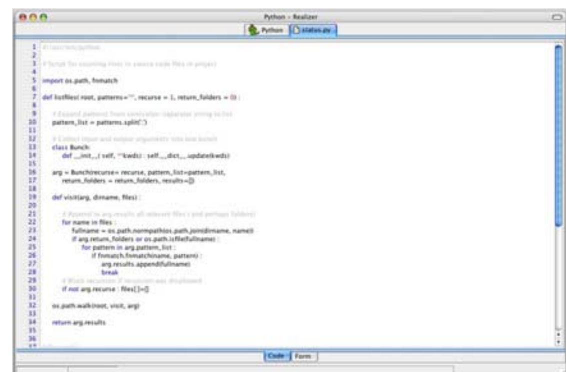
The editor component

The editor component is simply a sub classed *QTextEdit* class from the Qt framework.

I have connected my custom color syntax subsystem to it for the coloring of all the text shown in text editor.

The only method necessary to overload, is the key press event handler, which must later handle all kind of fancy editing functionality. Like automatically control the indentation levels in the source code. Bring out the list of methods in Python classes for easy selection and so one.

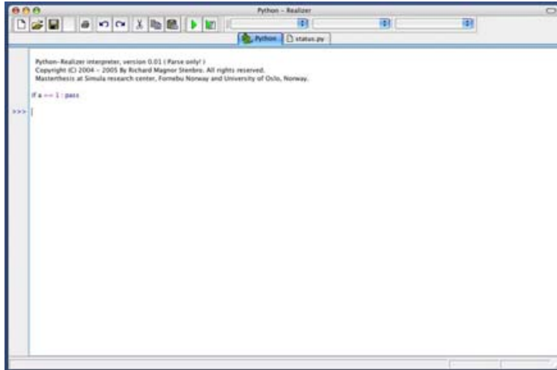
Later we will add a full form editor sub system to the editor as a separate tabulator. It is just a dummy form editor for now.



The interpreter component

The interpreter component is more complex than the editor component.

It uses the same syntax color system as the editor components do. I will therefore not comment on that, except for a little turn off coloring system, the interpreter use.



When the interpreter writes output from Python system to the user, it will not have this output colored as the standard user input. This is simply implemented as a pre text to all the output in form of a "! " combination. This turns off all the coloring for that given line only, and will not be shown on the screen.

This component also uses a *QTextEdit* class from Qt framework, and sub class it to overload all the key input from the user. Here it is much more important to control the user use of keyboard, since it is going to be sent to the parser in a timely fashion.

All communication between the interpreter window and the interpreter system is sent over a local TCP/IP socket connection on socket 8101.

This socket communication is also a communication line between the two threads used to run the whole system.

To implement this important socket connection I use two classes from the Qt framework. It is the *QSocket* class and the *QServerSocket* class.

In my design, the interpreter window is the client part of the connections, and it will ask for a valid connection by calling a server socket at port 8101. It will then receive a communication socket to use for the rest of the applications run time.

The Python interpreter system is designed to be a server in this communication link, and will only accept one client to connect at a time.

Later I have plans for yet another client / server communication link for the communicating of variable status, and the methods of the Python classes and so one. This is not yet planned, and maybe there is a better way to do this.

There still are a few bugs in the application that needs to be addressed before it can be found useful. One of the small annoyances is line numbers in the editor windows that still doesn't exactly cooperate at all times.

Since I am under a strict deadline, and since this is my second attempt to create a useful mockup of how I want the final application to look, I haven't prioritized bug killing.

Python parser in C++

Computers have problems understanding the humans and their languages directly, both orally and in the written form.

Humans can not easy or at all talk computer language directly, so we need an efficient way to translate automatically between what the humans understand and what the computers can execute for us.

All human languages is very complex and has a large number of different words, with different meaning based on the context of sentence or even the tone we speak the words in.

It is most likely impossible to make a system that can translate directly from a human language to something the computer can understand and then execute directly.

To complicate the process even further, we have several different instructions set on different computer system using different processors. This makes is mostly impossible to move programs between processors without having to translate again.

Even the operating system may differ so much, that a program can't just be moved from one system to another without seriously reprogramming.

To solve this problem in both ends of the translation process, it is normal to design special computer languages.

These are much simpler in their complexity and number of known words to handle. This makes it possible for smart programs like compilers or interpreters to translate from computer language designed for human understanding to something the computer can execute.

Since different computers use different processors with different instruction sets and architecture, we need to create separate translators for all the computer architectures we want to support.

Some systems are based on a complete different approach to solve this problem.

They use virtual byte code executable engines to execute a made up virtual instruction set that is decoded into real executable instructions.

You basically invent a virtual processor with a simple portable instruction set, and just write the translator layer for each real processor you want to execute on.

Your programs then, just need to target this virtual processor, and will be automatically available on all platforms with this translator.

The Python – Realizer system will eventually use such a system with a virtual executing machine and translating between Python code written into a text file or directly from user, and to this virtual instruction set, before finally being executed by translating to the native execute instruction set.

To ease the process of designing and understanding such a complex system as compiler and interpreter are, we need to divide and conquer it into several smaller components.

Each component is responsible for a limited part of the translation process and does that very efficient and

makes it easier to design and understand the whole system.

The first part of the translating process between the Python textural code and actual being able to executable on a designated destination platform is to read the actual text and translate it to what we call "Abstract Syntax Tree", or node tree which contains all necessary information needed to translate to the final virtual executable code.

The component responsible for this first translating step is called a parser. The parser is actually divided into two parts. The first part translates actual text word, numbers or other symbols to something the next step can handle.

This text is in the *Unicode*[13] format, and is capable of handling all the world alphabets in 16 or 32 bits chunks, instead of the old standard with 8 bits chunks and only 256 different characters.

The Result of this first translating process is a numerical representation of the textural symbol found in the text based source code. This is called tokens, which are often an *enum*[14] type with only legal symbols as member.

The second part is really the parser. It translates from such tokens based on the grammar rules into an abstract syntax tree with nodes for each of the language construct. This part is also responsible for reporting grammar errors found in the text.

All the information needed for each language construct, like the name of a variable or the text of a string, is stored into its corresponding node for future handling.

This part of the compiler process is what we call architecture in depended, but source language based.

This means it is locked to the source language, but can be used in several different compilers with different resulting executable code.

In our case, it is locked to Python 2.4 grammar, but can be used in several different compilers without having to modify the grammar or components made to handle it.

This part of the translation process is often referred to as the front-end stage of the translation, while the code generation phase is referred to as the back-end stage. In this chapter we will concentrate exclusive on the front-end phase of the translation process.

I will describe the design decisions made for the parser components and most of its details and its data structures.

The Parser is entirely written in C++ as a class with all the needed functions built-in as methods of that class. It consists of less than 3000 lines of C++ code, and is quite compact and easier to understand than most others.

I have made at least ten different version of this parser through the last year for different purposes. First I made several editions for the *STL*[15]

---

[13] Unicode – A standard for international character sets, supporting a larger set of characters than standard ascii set.

http://www.unicode.org

[14] Enumerating data type in C++

[15] Standard Template Library.

library and *wchar_t*[16] based characters.

When I decided to use Qt library exclusive instead of the STL library, I rewrote the parser for using data structures found in Qt like *QMap*[17] and *QValueList*[18] classes.

In other compiler designs and implementations, it is normal to use special tools to write the parser in, for automatically generation of parser source code. Such generators like Bison, Yacc, javaCC and others creates large and difficult to understand source code for the parser based on an input description of the grammar of the language.

It is often not so fast in its parsing duties either, and makes the error reporting more difficult to make sense of for the end user. It also means you need more tools available on all the platforms you want to support.

Even the all powerful gnu compiler suite, is in the process of converting its parser from these tools to straight C++ based handwritten parser.

One of my inspirations for the design of the new Python parser has been Microsoft's C# compiler[19], which is available in a open source edition for your own browsing.

It is very complex, since C# grammar is not very easy to translate directly without needing to know your context at all time. Python grammar is however a very well thought through and excellent designed for just one token look ahead at the time, for finding the right grammar rule to follow next.

We call this kind of parser, a recursive descent parser, which uses methods for each grammar rule it needs to travel in the process of parsing a source code file or user input.

You will find the latest edition of this C++ implemented parser in just two small files:

pythonParser.h
pythonParser.cpp

They are located in the following location at this location:

Kildekode/Realizer/Interpreter

Lexical analyzing

First step in the translation process from textural representation to an abstract syntax tree is to decode each textural element into something, the parser can understand. This is symbols called tokens and sometime data related to some of the tokens, like variable names and number contents.

It is important to know that, when I converted the parser component into pure Qt usage, I only needed to modify the understanding of this text format and switch the use of two small container classes.

The responsibility of the lexical analyzer, is to check for correct textural representation of variable name, correct numbers with collected result, legal Python operator use and reserved keyword found in grammar most be correctly collected.

[16] Wchar_t is a unicode character in C++.
[17] QMap is a dictionary type.
[18] QValueList is a list type.
[19] This was available on www.microsoft.com earlier. You may try to search for it on that web site. It is only the parser technique I used as an inspiration for my own parser.

The parser part will ask for one decoded token with collected data attached to it at the time, in the process of building the all important resulting node tree based on legal grammar rules.

The token type available to the parser from the lexical analyzer is grouped into three types.

The first group is the 29 different reserved keywords defined in the Python grammar with special meanings. These names can not be used as variable name, since that will confuse the parser when doing the translating.

Then we have a group of delimiters and operators. They are one to three letter long operator symbols like *+, -, ** and the likes.

And finally we have the literals like variable names, numbers and strings which need to be collected for the parser.

The Parser gets what it needs just by calling the same method over and over again until it gets an end of file marker. This method is called:

*TOKENID scanToken(FullToken *pFT)*

This method is the lexical analyzer, and returns information as a token code (TOKENID) and collected information in a data structure called FullToken.

Reserved keywords

The following reserved keywords are recognized in Python 2.4 grammar in my parser and in the original C based parser.

and	*assert*	*break*
class	*continue*	*def*
del	*elif*	*else*
except	*exec*	*finally*
for	*from*	*global*
if	*import*	*in*
is	*lambda*	*not*
or	*pass*	*print*
raise	*return*	*try*
while	*yield*	

These keywords can not be used as variable name, since they have special meanings in the grammar.

Operators

The following operators have special meaning in the Python grammar, and therefore needs to be used in the correct context.

```
+     -     *     **    /     //
%     <<    >>    &     |     ^
~     <     >     <=    >=    ==
!=    <>
```

Delimiters

The following tokens are used for separation of statements in the Python grammar.

```
(     )     [     ]     {     }
@     ,     :     .     `     =
;     +=    -=    *=    /=    //=
%=    &=    |=    ^=    >>=   <<=
**=
```

Special meaning tokens

Python also have four tokens with a very specialized meaning during parsing of the Python source code, they are:

<NEWLINE>
Indicates change of line in the source code, and will have different meaning based on its context.

<INDENT>
This is the way Python controls block of codes, by having different indentation levels for each unique code block. It does not use { } to control block structures like most other languages.

<DEDENT>
This is the matching token to indicate block end. A little like '}' in other languages.

<EOF>
This has two meanings in Python. In the process of parsing a file, it indicates that the end of file has been reached. In interactive mode, it just indicates that the parser may need more input from the user, or the end of this sentence.

Literals used in Python

Python has three literal types, which handles all the user defined data for the parser to generate necessary nodes in the abstract syntax tree.

They are as follows:

<NUMBER>
All valid numbers like 1.34 , 3J , 34L and others are decoded as token <NUMBER> with the needed information added as text string for later decoding.

<STRING>
String is all textural representation, which are started and ended with either one or three ' or " characters. Those with three can span several lines if needed, the one with only one can not span more than one line.

<NAME>
This is the name of the user variables, with collected name attached to the token for symbol table handling in later steps of code executing.

Implementation details

To speed up the process of checking for reserved keywords, every time a name token is found, we use a dictionary class for fast lookup of needed words.

In the Qt version of the parser, I use a simple

QMap< QString, TOKENID>

In this Qt class we use textural strings of the type *QString*[20] to store the lookup names, and the corresponding token symbol as return values.

When you have a text you want to check for keyword or name literal, you look it up in the dictionary and

[20] QString is Qt string class for all handling of text with Unicode or not format.

you get either the reserved token symbol if found or name symbol if not found.

The lexical analyzer is alone responsible for the block control in the parser, by giving the corresponding special token for indent, newline and dedent based on its context.

It makes sure we only have valid indents of code blocks in the whole program, and decides how many dedent(s) are needed when the code blocks end.

The token *<IDENT>* indicates a new code block, and the lexical analyzer needs to remember how many white spaces is collected before the code block start and store this for later check.

The Parser has no control of block levels, and really don't care about any thing else than getting <INDENT> token to indicate the new code block.

The token *<DEDENT>* can come in one or more symbols to the parser. Each token symbolize leaving one code block at a time in the grammar analyzing.

The lexical analyzer must make sure indentation levels are correct, or give error messages to the user.

The token *<NEWLINE>* is important to signal to the parser when a line breaks. Not all line breaks are given to the parser. If we have a line with only comments or white space, it is simply ignored and never sent to the parser as a token.

Example of block control and the corresponding token codes returned to the parser:

```
class test :

    # This is a test program!

    def __init__ (self, name):

        self.name = name
```

The resulting block control tokens returned to the parser will be:

... <NEWLINE>
Ignored three lines!
<INDENT> ... <NEWLINE>
Ignored line!
<INDENT> ... <NEWLINE>
<DEDENT> <DEDENT>

Every legal indentation levels are stored in a *QValueVector*[21] class structure declared as:

QValueVector<unsigned long>

It function as a vector for push and pop of the indentation levels as needed for correct issuing of the needed tokens to the parser for block control.

How many indentation values or levels we have pushed is stored in a simple variable called *mIndent*.

When we have a line break in the source code, a variable with the name *mPending* will control how many <DEDENT> tokens we will need to issue to the parser before it can continue analyzing the grammar rules.

[21] QValueVector is Qt vector class for push and pop of indentation levels.

The lexical analyzer is as we have described above, based on one method called *scanToken*.

This method returns token code to the parser, but it also take a pointer to a very important data structure called *fullToken*, which has space to store additional information about a token to be returned to the parser.

We will now present those data structures used in this method.

Data structure TOKENID

This is an enumeration type in C++ with values for all valid token a parser can get from the lexical analyzer. It is a numerical value for the parser, but a textural name for us.

All valid values are defined in the C++ header file mention earlier for the parser class, and I will not bore the readers with repeating it here.

It basically has the describing name for each Python token with data structure specific prename.

Examples are given below:

PY_EOF
End of file token

PY_NEWLINE
Line break token that count for the parser.

PY_FOR
Reserved keyword "for" token

It is a total of 80 different token codes defined and used in this parser, and that is describing all symbols used by the parser in Python 2.4 grammar.

By defining all tokens as member of an enumeration data type, we make sure that we can't send bogus tokens from lexical analyzer to the parser by mistake, and it is much easier to understand the textural names in stead of the plain numbers.

Data structure FULLTOKEN

All contact between the lexical analyzer and the parser is provided through *TOKENID* enumerations and if more details about each token found are needed, it is stored in a structure called *FULLTOKEN*, which have space for the information about line and column of start position of the token in text and optional collected data by some of the tokens.

This data structure is defined as:

```
typedef struct {
        TOKENID iToken;
        QString id;
        unsigned long iLine, iCol;
} FULLTOKEN;
```

I will now explain each data field used in this C++ structure.

The first field *iToken*, is just a copy of the returned token code from the lexical analyzer.

The second field is a *QString* for storage of text or number for name, string and number tokens.

The last field contains the line number and column index of that line, for the start character of the current token. This is for error reporting in parsing and executing of Python scripts both from file and user input.

Collecting text to analyze

The first few edition of my parser where designed to get its input from the standard input and report all output to the standard output or the standard error as needed. This made them easy to write and impossible to integrate in a GUI application in a good way.

Getting input from files was reasonably simple by using streams in C++ for reading characters into a buffer, containing characters in *wchar_t* Unicode format.

When I decided to only have a GUI interface to the interpreter and thereby the parser, I needed to rethink input / output system. Both reading input from files and directly from the user will be needed to be handled differently in a GUI environment.

As described in the application chapter, I have designed the system to use an internal TCP/IP communication on local sockets for all communication between the interpreter and the GUI of the application.

I use two separate threads in the application to divide the work between the GUI work, and the interpreting of Python code. This is to prevent freezing of user input in the application, when the interpreter is running.

I have also a locking mechanism to prevent user input from the interpreter window, when running Python code from a file or from the editor's memory.

This means that the parser needs to communicate both ways over a socket based connections to the GUI application's interpreter window.

All network communication internal in the application is handled by Qt's excellent support for Sockets. I use two Qt classes for that purpose. They are *QSocket* and *QServerSocket*.

I also use a class *QTextStream* connected to the sockets for streaming purposes.

When there is input from a file, I read the whole file into a QString, and access one character at the time from this class. When the input is supposed to be interactive directly from the user input, I request data from the socket and sleep until something arrives for me to process.

The interpreter system will send information to the interpreter window with request for input by showing a prompt.

There are two possible request prompts, and they are either >>> or if line continues on next line ...

We also have an internal prompt which is not seen in interpreter window for overriding text coloring in window of output text from the parser. We simply use a string with "!
" at the beginning of the line for this to work.

My parser, use some variables to control its work. We have counters to control the current place in buffer

and where the current line starts in the buffer. This is for local calculation of the line and column, and for knowing where to start looking for the next token.

We also have status flag for error situation and for controlling where we are getting our input, file or user input.

Abstract syntax tree

It is the parsers job to create a tree with all the necessary nodes in correct place and pointing to each nodes parent node as needed. When it does this job, it also checks for the correct syntax and collect necessary data to some of the nodes as needed.

Only if there is not found errors during building of this tree, will the code generator continue to do its work.

All the data structures that are not in this node tree will at end of the parsing become deleted. Only the final node tree is used by the next step in the translation done in a Python compiler.

Definition of node structure

Every node used in the abstract syntax tree is defined as a C++ structure as follows:

```
typedef struct Node {

        NODETYPE mType;

        NODE *pLeft, *pRight;

        QValueVector<NODE *>
m_siblings;

        unsigned long mLine, mCol;

        QString text;

} NODE;
```

This node structure contains all the information the compiler needs to generate code in a code generator, and all the information that will be inserted in a symbol table at runtime. This makes the parser a separate component that execute in depended of the rest of system, and it leaves it to the next step to continue the translation process.

It is important to notice, that the symbol tables will be used both under the code generation and under the executing of compiled byte code in the virtual machine, and not only in the first step as in traditional compilers.

Now I will describe each field of the node structure:

The first field is the one describing what kind of node we are, for the code generator to generate correct code for each of the construct in the tree.

This is also an enumeration type defined later, and does for the nodes what *TOKENID* does for the lexical analyzer.

Since we are designing a node tree, we need a pointer to left and right child node. This is the second field of the node structure. We could manage to build a tree with only this two child node to each parent node, but that will create larger tree than needed, and confuse us under the code generation phase.

We want to use a more compact node tree for easier code generation in a later step of the compiler, and easier understanding of Pythons construct to code conversion. To manage to compact the tree a lot, we collect more child nodes and put them into a vector. For this we have a *QValueVector* class container as third field in the node structure.

Not all the node types use more than two child nodes. Some even use none or just one child node to get its business done.

Because of possible error under the executing or code generating, we will need to store information about location for this node as line, column pare of where its token where found in the source code.

And last we store the text or the number found as part of the name, string or number nodes in text field.

It will be the code generator that will handle the decoding of this information on the construction of necessary Python data type objects.

Definition of node type

The parser uses a total of 93 different and unique node types to express the Python source code as abstract syntax tree. The code generator part of the Python compiler will work entirely on this tree to generate the finished code and data objects.

We will mainly divide node types into four groups. First we have the statement nodes to construct instructions for loops and jumps in the generated code based on some value calculated or found.

The parser also contains nodes for importing of other modules to be used in the context of this module.

A code example for this kind of node is as follows:

```
If a == 1 : pass
```

This will execute the pass statement based on whether a value 'a' will be equal to 1, else it will not.

This will be generating an if-node as the top parent with the equal-node as left child node and the pass-node as right child node. The equal-node will have a name-node as its left child node and a number-node as its left child node.

The name of the variable 'a' and the number '1' will be stored as extra information in the name and the number nodes.

The next group of node-types is expressions, which are calculations and data manipulating instructions. A lot of a programs work is done with these nodes / instructions.

Example of an expression node:

```
Test = a * 4 + ( b - 4 ) / 1.4
```

First node of this expression will be a assign-node (=) with a name node as left child node and rest of expression as follows on right child node.

The third group of node types is the container nodes for class, methods, functions and more. They also collect statements to be executed

continuously one after another as stored in the container node.

The last group is a very important one. It contains the literal nodes like name, string, number or more complex data collecting like dictionary and lists.

```
Test

0.34J

"Hello strings"

{ "a" : 1, "b" : 2 }

[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 ]
```

We will now present all node types in use by the parser in the node tree.

Statement nodes

NK_IF	NK_FOR
NK_DEF	NK_CONTINUE
NK_YIELD	NK_GLOBAL
NK_ELIF	NK_TRY
NK_CLASS	NK_RAISE
NK_EXEC	NK_FROM
NK_ELSE	NK_EXCEPT
NK_PASS	NK_BREAK
NK_PRINT	NK_IMPORT
NK_WHILE	NK_FINALLY
NK_DEL	NK_RETURN
NK_ASSERT	NK_LAMBDA

This are both complex and simple operations, one needs to generate code for, but often it is generated as little as one instruction based on those nodes.

Expression nodes

The following nodes are for expression described in the node tree:

```
NK_ASSIGN
NK_MINUS_ASSIGN
NK_DIV_ASSIGN
NK_BITWISE_AND_ASSIGN
NK_BITWISE_XOR_ASSIGN
NK_SHIFT_RIGHT_ASSIGN
NK_DOUBLE_DIV_ASSIGN
NK_ANDTEST
NK_LESS
NK_EQUAL
NK_LESS_EQUAL
NK_IN
NK_IS
NK_IS_NOT
NK_BITWISE_XOR
NK_SHIFT_LEFT
NK_PLUS
NK_MUL
NK_DOUBLE_DIV
NK_PREPLUS
NK_TILDE
NK_SUBSCRIPTLIST
NK_LISTFOR
NK_GENFOR
NK_RANGE
NK_PLUS_ASSIGN
NK_MUL_ASSIGN
NK_MODULO_ASSIGN
NK_BITWISE_OR_ASSIGN
NK_SHIFT_LEFT_ASSIGN
NK_POWER_ASSIGN
NK_ORTEST
NK_NOTTEST
NK_GREATER
NK_GREATER_EQUAL
NK_NOT_EQUAL
NK_NOT
NK_NOT_IN
NK_BITWISE_OR
NK_BITWISE_AND
NK_SHIFT_RIGHT
NK_MINUS
NK_DIV
NK_MODULO
NK_PREMINUS
NK_POWER
```

```
NK_SLICEOP
NK_LISTIF
NK_GENIF
```

Those node type, work with the users defined data structures and are the foundation for most of a programs work.

Container nodes

The following nodes are container nodes which are collecting statement nodes as member nodes.

```
NK_MODULE
NK_VARARGLIST
NK_SIMPLESTMT
NK_EXPRLIST
NK_DOTTED_AS_NAME
NK_DOTTED_NAME
NK_CONTAINER
NK_FORMALLIST
NK_TESTLIST
NK_IMPORT_AS_NAME
NK_DOTTED_AS_NAME
NK_IMPORT_AS_NAME
```

Literal nodes

The following nodes have always some kind of additional information for the code generation like name or number details.

```
NK_NAME
NK_LIST
NK_DICTENTRY
NK_NUMBER
NK_LISTGEN
NK_STRING
NK_DICTIONARY
```

Example node generation

I will now demonstrate a node tree building of a simple compound statement structure, using if / elif and else statements, and what kind of nodes there will be generated and in what order.

```
If a == 1 :
        print "Hello World!"

elif a == 2 :
        print "Bye World!"

else :
        pass
```

The parent node for this statement will be an if-node with the expression a == 1 as left child node. Every thing after colon in the first sentence will be placed as a right child node.

All "elif" statements and the one optional "else" statement will be placed in the siblings vector as they are discovered through the parsing.

Left node will be an equal-node with a name node on its left side and a number node on its right side.

On the right side of the parent node there will be a print-node with a string node as its left child node.

The rest of the statement sequence will be generated in much the same way, except for being an elif-node or else-node and put into the siblings vector in the order found. That is else-node will always be stored last.

Parser

At last we will detail the actual parser part of the parser. That is the grammar rules Python is built out of.

The parser is constructed as a series of methods, which each are handling checking a grammar sentence, and the necessary node generation to represent a grammar rule in the syntax tree.
We can delegate those methods into three groups of grammar rules.

We have the statements rules which represent grammar rules for the program control and organisation.

Then we have the expression grammar rules for the actual computation done in a program. This is also the group that manipulates the user data objects.

The last group is start rules for the grammar based on what we are supposed to parse into an abstract syntax tree with the corresponding nodes.

Do we parse from the user input or from a file are decided by choosing the right start grammar rule.

Start from user input

When doing interactive parsing from the user, we will let the parser ask for user input directly from the user as needed. The parser object must be created and started in asking for input mode. That means it is waiting for the input from a user over the local socket connections.

We will of course give the user necessary prompt when we ask for the input, to let him know what is expected by the parser at all times.

The parser starts by the following methods gets called by the interpreter in interactive mode:

Bool pythonParseFromUser
(QString lineText);

This provides the first line to be analyzed by the parser. This will start the parser by calling the following grammar rule:

Void pythonSingleInput();

We will now parse the user input and ask for more input as needed to complete all the grammar rules started by the user input or it will generate an error messages.

The complete grammar rule set, is provided in appendix A for completeness.

Start from file or buffer

My last edition of parser is capable of parsing both out of a file input and straight out of the editor buffer of the application.

To start parsing from a file, we start by calling the following start method:

Bool pythonParseFromFile
(const QString filename);

We will then try to open a file with the give filename and read the whole file into a QString buffer.

We start parsing of the top grammar rule by calling the following start rule method:

Void pythonModule();

It will then try to parse the whole file or report error messages to the user input window in the application.

When we want to parse the input directly from the editor window of the application without saving to a file first, we send the whole buffer as a QString buffer to the following start method:

*Bool pythonParseFromString
(QString buff, QString name);*

This will start the parsing with the following start grammar rule in the method:

Void pythonModule();

This method takes care of the start grammar rule for a file and an editor window source code with the following grammar in *EBNF*[22] format.

File_input: (<NEWLINE> | stmt)* <EOF>

This mean that a file input has zero or more stmt rules or tokens *<NEWLINE>* followed by the end of file token.

Compound statements

Grammar rules of the type compound statement, is a really important language construct for telling the computer how to manage the executing of a program.

This is the way to enable part of the program to be executed more than once in loops, and control which choice a program can follow to get the result done.

This grammar rules often contain several lines of code to manage one grammar rule. That is, one has several choices to where the program should jump, based on some computation done as part of the sentence.

I will first present a very common and import grammar rule for executing choices based on some expressions.

It is (if / elif / else) construct, which follows the following grammar rule in parser:

```
If_stmt:
        <IF> test <COLON> suite
        ( <ELIF> test <COLON> suite)*
        [ <ELSE> <COLON> suite ]
```

This construct, have always on start sentence with an 'if' clausal and an expression test part, before one get to the suite of statement to be executed on a true result.

Such construct, can have none or more 'elif' clausal as optional executing path for the program to follow based on the result of the test expression.

There is an optional else clausal available to get a default executing path for the grammar rule.

All the test expression must result in a Boolean value, so that a program can make its decision on which path to follow.

This grammar rule is programmed in the method:

NODE pyIFstatement(void);*

[22] Extended Backus-Naur Formalism.
A language design to describe grammar of computer languages in a formal way.

This rule, generate a *NK_IF* node with necessary nodes attached for rest of the grammar rule.

The next compound grammar rule to be discussed, is loop construct with an optional else part.

It is the (while / else) statement, described in the following grammar rule used by the parser:

```
While_stmt:
        <WHILE> test <COLON> suite
        [ <ELSE> <COLON> suite ]
```

This is handled by the following method in the parser:

NODE pyWHILEstatement(void);*

Based on the result of a test expression, the suite of statements will be executed none or many time, and finally the else part will be run when while terminates its looping.

This rule generates a *NK_WHILE* node with necessary nodes attached for the rest of the grammar rule.

The third compound statement, is the (for / else) construct, which implements the following grammar rule:

```
For_stmt:
        <FOR> exprlist <IN> testlist
                <COLON> suite
        [ <ELSE> <COLON> suite ]
```

This construct iterate the "exprlist" grammar rule over a "testlist" range, and does execute the suite of statement each time.

The else part will be executed at the end of 'for' iteration process as a way out of the loop construct.

This is handled by the parsers method with the following signature:

NODE pyFORstatement(void);*

This rule, generate a NK_FOR node with all the necessary nodes attached for the rest of the grammar rule.

The last compound statement for choice of executing path is the exception handling construct with the following two grammar rules:

```
Try_stmt:
( <TRY> <COLON> suite
        ( except_clause <COLON>
                Suite )+
        [ <ELSE> <COLON> suite ]
| <TRY> <COLON> suite
        <FINALLY> <COLON> suite )

Except_clause:
        <EXCEPT> [ test [ <COMMA>
                Test ] ]
```

This is a bit messy grammar rule, but I will explain. There are in reality, two different ways to program the exception handling in Python. The simple one is a simple try / finally statement, where one tries to execute a suite of statement in a try block.

When that is finished, the code in the finally block, will be executed.

The other exception construct grammar-rule is more complex and has the same suite of statements to execute in a try block, but gives one or more exceptions options if an error is en counted under the executing of try block.

It also have an optional else part, which will be executed if no exception where catch.

The construct mention is being handled by the parsers method:

NODE pyExcept(void);*

This is for both grammar rules and this produce a *NK_TRY* node with the necessary child nodes attached as needed to describe the full rule.

In addition to mention compound statements which is for program executing control, we have some to collect other statements into a group for class declaration and function declaration both inside and outside of the class.

The first one and a very important one in object oriented programming is the data type class.

It is defined by the following, very easy at first sight, grammar rule:

```
Classdef:
        <CLASS> <NAME>
                [ '(' testlist ')' ]
                        <COLON> suite
```

This is a construct for construction of a class definition, with the optional inheritance of one or more base classes, before it describe methods and local variables in the class type.

This is handled mainly by the following parser method:

NODE pyCLASSstatement(void);*

This rule generates a *NK_CLASS* node with its necessary nodes attached to describe rest of the grammar rule.

Last we have the function construct, used both inside and outside of class objects.

The grammar rule for them both, are the same simple grammar rule:

```
Funcdef:
        [ decorators ] <DEF> <NAME>
                '(' vararglist ')'
                <COLON> suite
```

This is handled by a total of four methods in parser:

NODE pyDEFstatement(void);*
NODE pyDecorator(void);*
NODE pyParameters(void);*
NODE pyVarArgsList(void);*

This rule, generate a NK_DEF node with its necessary nodes attached for the rest of the four grammar rules.

Decorators are a new construct addition to a function, new in Python 2.4 grammar, and it is included in my parser.

Simple statements

Python has a lot of different types of so called simple statements. That is statements which can be put on a single line, and often share a single line with only a ';' as delimiter. List of all the simple statements is found in grammar file as appendix A.

My Python parser takes care of all the simple statements in the following methods in the parser class:

NODE pyExprstmt(void);*

NODE pyPrintstmt(void);*

*NODE*pyDelstmt(void);*

NODE pyPassstmt(void);*

NODE pyBreakstmt(void);*

NODE pyContinuestmt(void);*

NODE pyReturnstmt(void);*

NODE pyRaisestmt(void);*

NODE pyYieldstmt(void);*

NODE pyGlobalstmt(void);*

NODE pyExecstmt(void);*

NODE pyAssertstmt(void);*

NODE pyImportstmt(void);*

Some of the simple statements can only be used in loops as a way out of a loop, or to jump to the next iteration.

These statements are break or continue. The return statement can only be used to return with or without values from a function or a method of a given class.

The pass statement is just for letting the Python interpreter a chance to ignore this code path under the executing, with do nothing as the actual meaning of this statement.

To import external modules for added functionality in the current script, we have several possible import statement constructs in Python.

All of the other simple statements can be used freely in the program, even as multiple single statements on the same line, separated only by a ';' character.

Simple statements grammar rule follows the following grammar rules in parser:

```
Expr_list:
        testlist ( augassign testlist |
        ( testlist '=' testlist )* )

Where augassign is one of:

        +=    -=    *=    /=    %=
        &=    |=    ^=    <<=   >>=
        **=   //=

Print_stmt:
        <PRINT> ( [ test ( <COMMA>
        test )* [ <COMMA> ] ]
|       '>>' test [ <COMMA> test )+
        [ <COMMA> ] ] )

Del_stmt:
        <DEL> exprlist

Pass_stmt:
        <PASS>

Break_stmt:
        <BREAK>

Continue_stmt:
        <CONTINUE>

Return_stmt:
        <RETURN> [ testlist ]

Yield_stmt:
        <YIELD> testlist

Raise_stmt:
        <RAISE> [ test [ <COMMA> test
        [ <COMMA> test ] ] ]

Import_stmt:
        Import_name | import_from

Import_name:
        <IMPORT> dotted_as_names

Import_from:
        <FROM> dotted_name
        <IMPORT>
        ( '*' | '(' import_as_name ')'
        | import_as_name )
```

Global_stmt:
<GLOBAL> <NAME> [<COMMA> <NAME>)*

Exec_stmt:
<EXEC> expr [<IN> test [<COMMA> test]]

Assert_stmt:
<ASSERT> test [<COMMA> test]

A simple explanation about the notation used in the grammar rule:

<NAME> is a token.
(|) is a two way grammar rule. Use either the one rule before '|' or the one after.

[ ] is an optional part of the grammar rule.

For zero or more occurrences of the grammar rule, we use the '*' character.

For one or more times the grammar rule, we use the '+' character.

We use '(' and ')' to group the grammar rule. This is of course much the same as regular expression matching.

The text before ':' character, is only the name for this particular grammar rule. This makes it easy to call the grammar rule from other rules.

Expressions

The last part of the set of all the grammar rules, but the one that do all the computation in a program, is expression statements.

It is usually the test grammar rule that start the decent into the expression rules until the parser found a legal data type or report an error messages trying to find one, to the user.

The way "down" in this method group is ordered by the following needed rules for operator precedence. I will end the explanation of the parser, with a guided tour down the descending parser in the expression statements.

All expression starts with a test rule, or a list of such rules. The parser starts with the following method:

NODE pyTest(void);*

In this method we take care of checking for the operator "or" or a lambda expression. The method will generate a node of type *NK_OR* or *NK_LAMBDA* if it found what it is looking for in this method, or else just returned the lower methods node upwards in the call tree.

Whenever it doesn't find the grammar rule, it will call the next method in descending order until it finally either find a valid token, or report an error.

Next method called, is the following method in the parser:

NODE pyAndTest(void);*

This method checks for operator "and", and generates a *NK_AND* node if found, else it calls the next method which is the following method in the parser:

NODE pyNotTest(void);*

Here we check for the operator "not" and we generate a NK_NOT node if successful. It is important to notice, that all this operators can be in series

of the same operator. Like the following expression with and operators:

A and b and c and d

This just generates "and" nodes attached to the one in front until all are generated into the syntax tree.

After the "not" operator, we will call a method who is handling the comparison between things. Things being a left and a right node expression compared with this operator.

This is taken care of in a single method called:

NODE pyComparison(void);*

This can check for the following comparison operators:

<	>	==	>=	<=	<>
!=	in	not in		is	is not

This will, if found, generate a node of one of the following types:

NK_LESS
NK_GREATER
NK_EQUAL
NK_LESS_EQUAL
NK_GREATER_EQUAL
NK_NOT_EQUAL
NK_IN
NK_NOTIN
NK_IS
NK_ISNOT

The next method to be called is the following method in the parser:

NODE pyExpr(void);*

This method can also be called directly from a method for a list of expressions.

We are now into the realm of bitwise operators. That is, operators working on bits of data and not on logical program jumps, as above methods have been handling.

It is the operator '|' that is being handled by generating a *NK_BITWISE_OR* node if found, or we call the next method in the parser:

NODE pyXorExpr(void);*

This method checks for existing of "xor" operator and generates a *NK_BITWISE_XOR* if successful, or call the next method in the parser:

NODE pyAndExpr(void);*

This checks for bitwise "and" operator and generates a *NK_BITWISE_AND* node if successful or else call the next method in the parser:

*NODE *pyShiftExpr(void);*

This method checks for one of two possible shift operators. That is, either "<<" or ">>" operator and generates on of *NK_SHIFT_LEFT* or *NK_SHIFT_RIGHT* node as a result.

Then we call the next method in the parser on our way down the expression call tree, which is:

NODE pyArithExpr(void);*

This checks for plus or minus operator, and will if successful return a *NK_PLUS* or *NK_MINUS* node to its caller.

The next method in the parser to be called on our way downwards is:

NODE pyTermExpr(void);*

This one, checks for one of the four possible operators, which are:

```
*    /    %    //
```

Generated nodes can be one of the following, if successful:

NK_MUL
NK_DIV
NK_MODULO
NK_DOUBLE_DIV

Next method to be called is:

*NODE *pyFactor(void);*

This method of the parser, simply checks for possible plus, minus or the symbol "~" tokens, as a pre operator to the rest of the expression.

Nodes generated are of one of the following possible node types if successful:

NK_PREPLUS
NK_PREMINUS
NK_TILDE

Next method is:

NODE pyPower(void);*

This methods just check for the '**' operator, which result successfully in a *NK_POWER* node.

Then finally, we get down to the atoms in the very important method called just:

NODE pyAtom(void);*

Here we check for the all important "atoms" of Python and collect the necessary information about the found data type.

Atoms result in nodes of the type string, name or number. It can also be collecting more complex data structures in Python as list, tuple and dictionary.

To get a full overview of all the grammar rules and their place in the parser, please see Appendix A for full details of the grammar of Python 2.4

Data structures in Parser

For the parser part of the parser class, we use only a few variables to control the executing of the parse process. These are as following:

bool isCompound;

This indicates when the parser is parsing a compound statement, which needs a little more attention to the details than simple statements.

bool isError;

This one is simple a flag for reporting error found during the parsing or the tokenizing of source code, so we don't do unnecessary work in the translation process.

unsigned long m_loops;

This one simply takes care of how many levels of loops we are currently run into during the parsing of statements. This is needed for controlling ways out of a loop.

unsigned long m_returns;

This one keeps order of how many methods or function we have to return

from before entering the top level of a program.

I suggest you study the parsers source code for the full understanding of the parser process. This is about the tenth edition of my parser. Older version is still available on my net site, as python_old.tar.gz

The Python type system

I am implementing the type system for Python - Realizer (PR) as a class framework in a dynamic loadable library called

libPythonBuiltinsLibrary

This will have different endings based on which platform it is compiled for. It will have a "DLL" ending on the windows platforms, "SO" ending on a Linux platform and "dylib" on the Macintosh platform.

This object library is designed for both to be used directly by the PR application and separately as a C++ class library if needed in other applications.

It even has C++ operators built-in in some of the data type, for more efficient use in C++ based programs.

The class framework is based on a base class built for garbage collecting all memory allocated for objects in Python. I have chosen to use a pre existing garbage system called "gc", which is available as a pre build dynamic loadable library.

It is important to now, that I have not yet started to use this garbage collector, and it is not implemented as the top base class yet. This to make sure my project builds on all platforms without pre installing too many depended systems.

This garbage collector is *Hans Boehm's Garbage collector*[23].

[23] More information and source code is available at www.hpl.hp.com/personal/Hans_Boehm/gc/

To start using this garbage collector, you just need to let the *PY_Object* class inherit the base class gc. After this, every call to the "new" operator will give you garbage collected memory to hold the object in use.

Every data type in Python - Realizer has *PY_Object* as a base class, and all common functionality will be stored in it. The real type system is a enumeration called *PYTHON_TYPE* , which has member types in the form of TYPE_xxx , where xxx is the name of the type in capital letters.

You interrogate the object about its type by calling the base class method *__type__()* which returns its type.

Later I will implement C++ operators in this base class to automatically support valid operators on all Python - Realizer objects. For now there is operator overloading implemented in some of the data types, and they get listed where needed.

All general references to Python objects get by a pointer to its base type. That is not the garbage collector class.

I will now explain each data type in details as planned to be implemented and what is started to be implemented already.

First I will bring to the attention of the reader, that there is a few data type supported in standard Python that is not planed to be implemented in Python - Realizer.

The first is the class for Unicode strings and characters. Since we always use Unicode in all the strings and character handling, we do not need a data type just for Unicode

added to the standard string data type.

Later we can opt for implementing the missing data types as needed to support both C and C++ extensions to our type system.

We do bring a special data type usually buried into the compiler, out to the standard data type library for general purpose use if needed. That is the code object, which is a container for the Python's virtual executing engine op codes.

It is not expected that users will make such code objects by hand, since it is still the duties of the compiler to do that automatically.

This library is in itself considerable amount of work, and is by no means finished. It is just started to be implemented. All classes are present for documentation purposes, but only a few have been started implemented and none is finished.

All source code for this data type system is available in the following path:

kildekode/Realizer/PythonLibrary

Both the header and the code files are listed here.

For extensive documentation of the C API used in standard Python and that needs to be implemented at some stage in the development of the C++ version, is found in api[24] document as part of Pythons online documentation found at www.python.org

For the documentation about syntax and meaning of Pythons special methods, you may read Python in a nutshell page 90 – 99 for the details.

The original C edition of Python have all its data types defined in the include directory and in the Object sub directory as simple *.h or *.c files. This is where I got most of the information needed for an understanding of the standard C edition of Pythons type system.

In my C++ version of Python, I have all type related files placed in the PythonLibrary directory. Here I place all the header files and the source code files for easy access during the development phase.

Abstract data type

This is an abstract class for access to external data types with Pythons interface for access to those methods being exported from such classes.

It is not started to be designed or implemented at this time.

Details about the standard C edition of Python implementation of this data type is available in the header file *abstract.h* and in the source code *abstract.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_AbstractObject.h* and in the source code *PY_AbstractObject.cpp*

Boolean data type

This data type is implemented as two static objects in standard Python. One is for true and one is for false. All use

[24] API – Application program interface.

of these two states in programs, just point to one of these.

I have however chosen to implement this data type as a dynamic class that is instantiated each time it is need to use one. This is mainly for homogeneous implementation of all data types and for use in C++ programs as well.

The data for this class is just a Boolean member value in the class called *mValue*, which stores true or false states.

This class has a few C++ overloaded operators available for use. These are as following:

+	&	/	(float)	(int)
(long)	%	*	|	<<	>>
-	^			

Some of these operators are for type conversion to float, int or long from the Boolean format.

This data types support the following special methods:

__abs__	__add__
__and__	__class__
__cmp__	__coerce__
__delattr__	__div__
__divmod__	__doc__
__float__	__floordiv__
__getattribute__	__getnewargs__
__hash__	__hex__
__init__	__int__
__invert__	__long__
__lshift__	__mod__
__mul__	__neg__
__new__	__nonzero__
__oct__	__or__
__pos__	__pow__
__radd__	__rand__
__rdiv__	__rdivmod__
__reduce__	__reduce_ex__
__repr__	__rfloordiv__
__rlshift__	__rmod__
__rmul__	__ror__
__rpow__	__rrshift__
__rshift__	__rsub__
__rtruediv__	__rxor__
__setattr__	__str__
__sub__	__trudiv__
__xor__	

This data type is still missing a lot of its functionality.

Details about the standard C edition of Python implementation of this data type is available in the header file *boolobject.h* and in the source code *boolobject.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_BoolObject.h* and in the source code *PY_BoolObject.cpp*

Buffer data type

This is a data type for access to buffers of bytes in a standard fashion.

It is not started to be designed or implemented yet, but it is planned to be extended with the following C++ overloaded operators for access to buffer data:

[]

Buffer data will be access by this these constructs:

Item = bufferobject[index];

And set by:

Bufferobject[index] = data;

Details about the standard C edition of the Python implementation of this data type is available in the header file *bufferobject.h* and in the source code *bufferobject.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_BufferObject.h* and in the source code *PY_BufferObject.cpp*

Cell data type

This data type is for cell handling with just two possible methods:

__get__	__set__

This is for accessing cell information or for setting new data into cell.

It will most likely be extended with the following C++ overloaded operators.

[]

This is for the same functionality as the methods defined in the class, but in a C++ way for use in C++ programs and in the Python system.

This is not started to be designed or implemented at this time.

Details about the standard C edition of Python implementation of this data type is available in the header file cellobject.*h* and in the source code cellobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_CellObject.h* and in the source code *PY_CellObject.cpp*

Class data type

This class is for all handling of Python based class. It will contain some form of list or table for the member methods and the variables connected to this class.
It also must contain a table of all of its base classes, it is supposed to inherit.

This is not yet implemented.

Details about the standard C edition of Python implementation of this data type is available in the header file *classobject.h* and in the source code classobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_ClassObject.h* and in the source code *PY_ClassObject.cpp*

Code data type

This is not a user specific data type at all, and not part of the built-in library of standard Python. It is in its compiler.

I have chosen to implement it in the built-in data type library for completeness and for use by the C++ programs directly if needed.

It is mainly a container for op codes in an array, and storage for local variables and constants. It is also heavily connected to the compiler and of course the virtual machine.

This data type is started, and has support for the following C++ overloaded operators:

[]

It is meant for code access under executing, and for code inserting by the compiler during the translation phase or reading in from pre compiled files.

It still remains a lot of design and implementation work on this data type, as the virtual machine and compiler gets implemented.

Instructions will be executed by accessing the code in the following manner:

Codeobject[instruction_ptr++]

This will return the current instruction to be executed and the instructions pointer is incremented for next access.

Details about the standard C edition of Python implementation of this data type is available in the header file *compiler.h* and in the source code *compiler.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_CodeObject.h* and in the source code *PY_CodeObject.cpp*

Complex data type

This data type is for handling so called complex numbers. That is numbers with both a real and an imaginary part.

It is stored in the class as two double variables. One is for the real part, and the other is for storing the imaginary part.

It will be extended with the following C++ overloaded operators:

+	/	==	>=	>	(int)
<=	<	%	*	!=	-

It also has the following methods for manipulating and controlling the data type:

conjugate
imag
real

It also supports the following special methods for this data type:

__abs__	__add__
__class__	__coerce__
__delattr__	__div__
__divmod__	__doc__
__eq__	__float__
__floordiv__	__ge__
__getattribute__	__getnewargs__
__gt__	__hash__
__init__	__int__
__le__	__long__
__lt__	__mod__
__mul__	__ne__
__neg__	__new__
__nonzero__	__pos__
__pow__	__radd__
__rdiv__	__rdivmod__
__reduce__	__reduce_ex__
__repr__	__rfloordiv__
__rmod__	__rmul__
__rpow__	__rsub__
__rtruediv__	__setattr__
__str__	__sub__
__truediv__	

It is only partly implemented at this time, and still misses a lot of functionality.

Details about the standard C edition of Python implementation of this data type is available in the header file *complexobject.h* and in the source code complexobject*.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_ComplexObject.h* and in the source code *PY_ComplexObject.cpp*

Description data type

This is a data type for description of external class methods and access to variable members in that class.

This is not yet been designed or implemented at this time.

Details about the standard C edition of Python implementation of this data type is available in the header file descrobject*.h* and in the source code descrobject*.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_DescrObject.h* and in the source code *PY_DescrObject.cpp*

Dictionary data type

This data type is for dictionary storage of objects. You find data items stored in dictionary by providing a key.

It will be stored internally in class in a Qt framework dictionary:

QMap<PY_Object *, PY_Object *>

It can use general PY_Object pointer that can return a hash value as a key, and every kind of data objects as data in dictionary.

It will be extended by at least these C++ overloaded operators:

```
==    >=    >    <=    <    !=
[]
```

With this you can lookup data with the following construct:

Data = dictionary[key];

And enter new data with the following construct:

Dictionary[key] = data;

This data types has a few methods for manipulating and controlling the data type:

```
clear
copy
fromkeys
get
items
iteritems
iterkeys
itervalues
keys
pop
popitem
setdefault
update
values
```

It also contains the following special methods for dictionary operations:

```
__class__        __cmp__
__contains__     __delattr__
__doc__          __eq__
__ge__           __getattribute__
__getitem__      __gt__
__hash__         __init__
```

__iter__	__le__
__len__	__lt__
__ne__	__new__
__reduce__	__reduce_ex__
__repr__	__setattr__
__setitem__	__str__

This data type is started to be implemented but not finished. It will be haps be used in the symbol tables for variable look up, and storage for constants in the code objects and more.

Details about the standard C edition of Python implementation of this data type is available in the header file dictobject.*h* and in the source code dictobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_DictObject.h* and in the source code *PY_DictObject.cpp*

Enumeration data type

This is a data type for enumeration data storage. It is not yet designed or started implemented.

Details about the standard C edition of Python implementation of this data type is available in the header file enumobject.*h* and in the source code enumobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_EnumObject.h* and in the source code *PY_EnumObject.cpp*

File data type

This data type is for all reading or writing to files on discs. It is implemented by using a *QFile* as a member of this class.

It uses a *QTextStream* for the actual text management in file access.

This class has status bits for:

Read allowed.
Write allowed.
File is open.
Binary form expected.

The following methods are defined for manipulating of files:

close
closed
fileno
flush
isatty
mode
name
newlines
next
read
readinto
readline
readlines
seek
tell
truncate
write
writelines

This data type is mostly already implemented and ready for action. It will be used by other library modules with file access as a speciality.

It also uses the following special methods functions in Python:

__class__	__delattr__
__doc__	__getattribute__
__hash__	__init__
__iter__	__new__

__reduce__	__reduce_ex__
__repr__	__setattr__
__str__	

More details are found in data types source file.

Details about the standard C edition of Python implementation of this data type is available in the header file fileobject.*h* and in the source code fileobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_FileObject.h* and in the source code *PY_FileObject.cpp*

Float data type

This is the data type for all the floating point number handling in Python. It uses a simple double variable as a storage system for numbers.

I am extending this data type with the following C++ overloaded operators:

+	/	==	(float)		>=
>	(int)	<=	<	%	*
!=	-				

This data type does not have any methods for manipulating this data type, except for these special methods, which are available for the float data type:

__abs__	__add__
__class__	__coerce__
__delattr__	__div__
__divmod__	__doc__
__eq__	__float__
__floordiv__	__ge__
__gt__	__hash__
__init__	__int__
__le__	__long__
__lt__	__mod__
__mul__	__ne__
__neg__	__new__
__nonzero__	__pos__
__pow__	__radd__
__rdiv__	__rdivmod__
__reduce__	__reduce_ex__
__repr__	__rfloordiv__
__rmod__	__rmul__
__rpow__	__rsub__
__rtruediv__	__setattr__
__str__	__sub__
__truediv__	

The C++ overloaded operators is mostly comparison related, and for standard mathematically operations on number or for conversion between the number types.

Details about the standard C edition of Python implementation of this data type is available in the header file floatobject.*h* and in the source code floatobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_FloatObject.h* and in the source code *PY_FloatObject.cpp*

Frame data type

This is a context data type for the Pythons virtual execute engine. It contains context specific information for the code object just being executed.

It can be in executing of a function or a code block. It is not finished at all, and is heavily connected to the development of the virtual machine.

Users do not create this data type. It is only useful for the compiler and the virtual execute machine.

Details about the standard C edition of Python implementation of this data type is available in the header file frameobject.*h* and in the source code frameobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_FrameObject.h* and in the source code *PY_FrameObject.cpp*

Function data type

This is all procedures used outside of class, and thus called functions instead of methods.

This data type is not yet implemented, but it will most likely include these methods to operate on this data type:

func_closeure
func_code
func_defaults
func_dict
func_doc
func_globals
func_name

It will also support the following special methods:

__call__	__class__
__delattr__	__dict__
__doc__	__get__
__getattribute__	__hash__
__init__	__module__
__name__	__new__
__reduce__	__reduce_ex__
__repr__	__setattr__
__str__	

There will most likely not be any defined C++ overloaded operators on this data type.

Details about the standard C edition of Python implementation of this data type is available in the header file *funcobject.h* and in the source code funcobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_FuncObject.h* and in the source code *PY_FuncObject.cpp*

Generator data type

This data type is for supporting the generator functionality in Python 2.4 and upwards. It is not designed or implemented, but the grammatically rule for this is in place already as a part of the parser.

Details about the standard C edition of Python implementation of this data type is available in the header file genobject.*h* and in the source code genobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_GenObject.h* and in the source code *PY_GenObject.cpp*

Integer data type

This is a very much used data type in Python scripts. It is implemented internally as a long variable for storage of all normal integer numbers.

We also have a data type for handling of even bigger integer number, which will be described later in this chapter.

I have chosen to provide the following C++ overloaded operators for easier programming in C++ programs:

<table>
<tr><td>+</td><td>&</td><td>/</td><td>(float)</td><td></td><td>(int)</td></tr>
<tr><td>(long)</td><td><<</td><td>%</td><td>*</td><td>|</td><td>>></td></tr>
<tr><td>-</td><td>^</td><td></td><td></td><td></td><td></td></tr>
</table>

All of these operators are for either mathematically operations on this data type or conversion to other formats.

We do not have any methods for manipulating or operating on this data type, except for these special methods:

__abs__	__add__
__and__	__class__
__cmp__	__coerce__
__delattr__	__div__
__divmod__	__doc__
__float__	__floordiv__
__getattribute__	__getnewargs__
__hash__	__hex__
__init__	__int__
__invert__	__long__
__lshift__	__mod__
__mul__	__neg__
__new__	__nonzero__
__oct__	__or__
__pos__	__pow__
__radd__	__rand__
__rdiv__	__rdivmod__
__reduce__	__reduce_ex__
__repr__	__rfloordiv__
__rlshift__	__rsub__
__rtruediv__	__rxor__
__setattr__	__str__
__sub__	__truediv__
__xor__	

Details about the standard C edition of Python implementation of this data type is available in the header file intobject.*h* and in the source code intobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_IntObject.h* and in the source code *PY_IntObject.cpp*

Iterator data type

This data type is simply for supporting iteration over sequence data types such as list, tuple or strings.

You will with the help of this data type get members of the sequence data type in a series of simple items, which one then can manipulate one at a time repeatedly.

This data type is not designed or implemented yet, but is needed in several other data types shortly.

Details about the standard C edition of Python implementation of this data type is available in the header file iterobject.*h* and in the source code iterobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_IterObject.h* and in the source code *PY_IterObject.cpp*

List data type

This data type is a lot like the tuple data type described later. It is however not limited in size, and can be extended freely.

I use the same Qt framework container class for storage of list:

*QValueList<PY_Object *>*

I have extended the data type with the following C++ overloaded methods to aid usage of this class in C++ programs and in Python:

+	==	>=	>	+=	*=
<=	<	*	!=	[]	

Most of these are for comparison related purposes, but we have two not so common ones. These are the incremental equation operators, which we have two of:

List1 += List2;

*List3 *= List1;*

The list data type has a few methods for manipulation and operation on list objects. They are the following methods:

append
count
extend
index
insert
pop
remove
reverse
sort

These methods are self explained, but one can look into the source code for implementation details.

The list data type also has the following special methods function available:

__add__	__class__
__contains__	__delattr__
__delitem__	__delslice__
__doc__	__eq__
__ge__	__getattribute__
__getitem__	__getslice__
__gt__	__hash__
__iadd__	__imul__
__iter__	__le__
__len__	__lt__
__mul__	__ne__
__new__	__reduce__
__reduce_ex__	__repr__
__rmul__	__setattr__
__setitem__	__setslice__
__str__	

Full implementation details are available in the source code for this class.

Details about the standard C edition of Python implementation of this data type is available in the header file listobject.*h* and in the source code listobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_ListObject.h* and in the source code *PY_ListObject.cpp*

Long data type

This data type is expected in Python to manage infinite large integer numbers in the form of strings and do mathematically operations on these strings when one or more of the numbers are too big for the ordinary variables.

This is slow, so Python tries to use the format long long to store numbers and do calculations on them, when they are "small" enough to be stored in one.

On a 32-bits computer, this data type is normally if present, a 64 - bits integer.

In my mind, Python is slow enough, so slowing it down with this kind of calculation is not a good idea. I therefore opted for just implement it with double long integer storage, and there by limit it to 64-bits integer.

I have extended this data type with the following C++ overloaded operators to simply use it in C++ programs:

<table>
<tr><td>+</td><td>&</td><td>(int)</td><td>(long)</td><td></td><td><<</td></tr>
<tr><td>%</td><td>*</td><td>|</td><td>>></td><td>-</td><td>^</td></tr>
</table>

This data type doesn't have any methods for modifying or operations on it, but it has support for the following special methods:

__abs__	__add__
__and__	__class__
__cmp__	__coerce__
__delattr__	__getnewargs__
__hash__	__hex__
__init__	__int__
__invert__	__long__
__lshift__	__mod__
__mul__	__neg__
__new__	__nonzero__
__oct__	__or__
__pos__	__pow__
__radd__	__rand__
__rdiv__	__rdivmod__
__reduce__	__reduce_ex__
__repr__	__rfloordiv__
__rlshift__	__rmod__
__rmul__	__ror__
__rpow__	__rrshift__
__rshift__	__rsub__
__rtruediv__	__rxor__
__setattr__	__str__
__sub__	__truediv__
__xor__	

Operators declared for use on this data type is mostly for mathematically operations and conversion to other data types.

Details about the standard C edition of Python implementation of this data type is available in the header file longobject.*h* and in the source code longobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_LongObject.h* and in the source code *PY_LongObject.cpp*

Method data type

This data type is for functions used in classes, and is renamed "methods" for this use.

The class data type is not yet started to be designed or implemented, and this data type is very connected to that data type.

It will be designed and implemented at the same time and in cooperation with the development of the class data type.

Details about the standard C edition of Python implementation of this data type is available in the header file methodobject.*h* and in the source code methodobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_MethodObject.h* and in the source code *PY_MethodObject.cpp*

Module data type

Modules is a group of functions, classes and variables stored in a file as Python source code or as a pre

compiled binary file, for import into current scripts to use its functionality directly.

Modules can also be C or C++ programmed groups of functionality collected into a group for simple access in our own programs.

Even though the parser has support for declaring import statements for just this kind of work, I have not started to design or implement this data type at this time.

Details about the standard C edition of Python implementation of this data type is available in the header file moduleobject*.h* and in the source code moduleobject*.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_ModuleObject.h* and in the source code *PY_ModuleObject.cpp*

None data type

This is pretty much like a advanced edition of the C++ NULL pointer, but it is promoted to a separate class in PR's type class framework.

It is not really designed and not started to be implemented at this time. I have however made a class template for it.

Details about the standard C edition of Python implementation of this data type is available in the header file *object.h* and in the source code *object.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_Object.h* and in the source code *PY_Object.cpp*

Object base type

This is the base class for all other data types, and will in time inherit the garbage collector class for automatically support for dynamically deleting of objects.

It also controls the type system, with its *__type__()* method for inquiries about each objects type.

Later it will have some kind of support for error states, when objects are used solely in a C++ program without Pythons error mechanism.

I am considering implementing a few C++ overloaded operators at his level, to control use of operators in sub classes in C++ programs.

This data type is not a stand alone one, and is only intended to be inherited by a sub class.

It is used for general pointing to all valid sub classes, before we have a chance to inspect the type of the class and then cast it to the right pointer type.

It still needs a lot of work until it is considered finished.

Details about the standard C edition of Python implementation of this data type is available in the header file *object.h* and in the source code *object.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_Object.h* and in the source code *PY_Object.cpp*

Range data type

This data type is used heavily in for statements to define range of for loop to iterate over.

This simply defines a start and end point for looping and an optional step values between these markers.

It then generates the needed values for iteration in for statements.

It will be extended with the following C++ overloaded operators:

+	==	>=	[]	>	+=
*=	<=	<	*	!=	

This is mostly for mathematically operation on this data type or for comparing relations.

This data type has the following methods for manipulating and controlling of data types operations:

append
count
extend
index
insert
pop
remove
reverse
sort

And it supports the following special methods:

__add__	__class__
__contains__	__delattr__
__deliten__	__delslice__
__doc__	__eq__
__ge__	__getattribute__
__getitem__	__getslice__
__gt__	__hash__
__iadd__	__imul__
__init__	__iter__
__le__	__len__
__lt__	__mul__
__ne__	__new__
__reduce__	__reduce_ex__
__repr__	__setslice__
__str__	

I have not yet implemented this data type as of this writing, but it will be most likely in form of three variables containing start, end and the optional step value.

Details about the standard C edition of Python implementation of this data type is available in the header file rangeobject.*h* and in the source code rangeobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_RangeObject.h* and in the source code *PY_RangeObject.cpp*

Set data type

This is the data type for handling of sets. It is not yet designed or implemented at this time.

Details about the standard C edition of Python implementation of this data type is available in the header file setobject.*h* and in the source code setobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_SetObject.h* and in the source code *PY_SetObject.cpp*

Slice data type

This data type is a bit special for Python, which slices out data from other data types which supports sequence and slicing of its data.

Examples of where one would use a slice data structure as helper is:

List1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Sliced = List1[3, 5];

Sliced is now a new list with just two numbers:

[4, 5]

This data type is needed, but still not started to be designed or implemented at this time.

Details about the standard C edition of Python implementation of this data type is available in the header file sliceobject.*h* and in the source code sliceobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_SliceObject.h* and in the source code *PY_SliceObject.cpp*

String data type

This is a very important data type for all handling of strings and single characters in Python. It is implemented as a Unicode based container class for the storing of strings and operating on them via built-in methods and external API functions.

It stores it strings locally as a *QString* class, which is wrapped into this class and added with methods for manipulating this data structure indirectly, since it is a private member variable.

When possible, I will try to use *QString* class own methods to do the work needed by the Python methods on the string type.

As a C++ extension, I have implemented a few overloaded operator methods. They are the following:

+	==	>=	[]	>	<=
<	%	*	!=		

This class has a few methods to operate directly on strings. These are as following:

capitalize
center
count
decode
encode
endswith
expandtabs
find
index
isalnum
isalpha
isdigit
islower
isspace
istitle
isupper
join
ljust
lower
lstrip
replace
rfind
rindex
rjust
rstrip
splitlines
startswith
strip
swapcase

title
translate
upper
zfill

Most of these methods will be implemented, but those who work with different text encoding schemes will most likely be dropped since we use Unicode on all the strings.

String class also has a few special methods available for operating on this class type. These are as following:

__add__	__class__
__contains__	__delattr__
__doc__	__eq__
__ge__	__getattribute__
__getitem__	__getnewargs__
__getslice__	__gt__
__hash__	__init__
__le__	__len__
__lt__	__mod__
__mul__	__ne__
__new__	__reduce__
__reduce_ex__	__repr__
__rmod__	__rmul__
__setattr__	__str__

One very important C++ extension to this data type, is the access to string data through the [] operator. You can set data into a string with this statement:

String[3] = 'a';

And you can get single characters out of string by simply use the following statement:

Test = String[4];

Other operators on this class, is mostly for comparison between objects or addition or multiply of strings.

Details about the standard C edition of Python implementation of this data type is available in the header file stringobject*.h* and in the source code stringobject*.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_StringObject.h* and in the source code *PY_StringObject.cpp*

Struct data type

This is a data type for interfacing with external code written in other languages that uses structured data types for its computation.

When Python needs to work with the external functionality's need for structure to hold its data, this data type comes into play.

It is not designed or implemented at this time, and needs a little research before it will.

Details about the standard C edition of Python implementation of this data type is available in the header file structseq*.h* and in the source code structseq*.c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_StructObject.h* and in the source code *PY_StructObject.cpp*

Tuple data type

This data type is a bit special. In standard C edition of Python this data type is defined with an initial size at

the creation time and can not be resized after this. It can however alter already allocated items in the data structure.

I have however implemented this data type with Qt frameworks data type *QValueList<PY_Object *>* which can be resized easily.

We have to simulate this size restriction to be compatible with standard Python. At construction of this object, we set a fake size limit, which the data type, respect by inserting NULL pointers into list to fill the list to expected size.

Then it is not allowed to append any more items into the list. Just alter the NULL pointers into valid data items.

We have extended the tuple data type with the following C++ operator overloading methods:

+	==	>=	>	<=	<
*	!=	[]			

This class has also no methods for manipulating this data type, expect for the special methods defined for this data type. They are the following:

__add__	__class__
__contains__	__delattr__
__doc__	__eq__
__ge__	__getattribute__
__getitem__	__getnewargs__
__getslice__	__gt__
__hash__	__init__
__iter__	__le__
__len__	__lt__
__mul__	__ne__
__new__	__reduce__
__reduce_ex__	__repr__
__rmul__	__setattr_
__str__	

Most of the operators we have overloaded for this data type are in comparison related operations, and we have added the same get / set functionality as in string data type.

The plus operator takes two objects and creates a new as a sum of these two given objects.

Details about the standard C edition of Python implementation of this data type is available in the header file tupleobject.*h* and in the source code tupleobject.*c*

Details about the current implementation of the C++ version of this Python data type is available in the header file *PY_TupleObject.h* and in the source code *PY_TupleObject.cpp*

Get arguments system

Most methods defined in Python takes a tuple as a parameter container when methods get called.

All the methods needs to decode this tuple into its simple items, and converted to C++ data structures for easy manipulating of the given parameter(s).

To ease the conversion we have a few support functions call, implemented as static member of a class for the whole conversion process and a built in error reporting system.

You just give these methods a tuple as an argument, a format string with conversion information, and a variable list of destination variables. The rest of the job is automated by this subsystem.

The format strings syntax is as follows:

c char

A Python string of length one become a C++ char.

d double

A Python float becomes a C++ double.

D PY_Complex

A Python complex becomes a C++ PY_Complex.

f float

A Python float becomes a C++ float.

i int

A Python int becomes a C++ int

l long

A Python int becomes a C++ long

L long long

A Python long becomes a C++ long long or 64 bits integer.

O PY_Object

Get a reference to object in Python.

s string

Python string without embedded null becomes a C++ string. (QString)

(...) as per ...

A Python sequence is treated as one argument per item.

|

The following arguments are optional.

:

Format finished, followed by function name for error messages.

;

Format finished, followed by entire error message text.

There are more defined format characters in Python, but it is not yet considered for inclusion in this C++ version of the Python yet. Some may not be needed at all. For full details, read Python in a nutshell page 522.

Example of a format string might be as follows:

"ii|i:Testfunc"

This means simply that we have two integer objects followed by an optional integer object, and this is the Testfunc function if errors are en counted during decoding.

The four functions defined for decoding of arguments are only partial started to be implemented and still needs some work before they are useful.

They are as following:

PyArg_ParseTuple

PyArg_ParseTupleAndKeywords

PyArg_vaParseTuple

PyArg_vaParseTupleAndKeywords

Details about the current implementation of the C++ version of this Python data type is available in the header file *getargs.h* and in the source code *getargs.cpp*

Build data objects system

We also have a function for building Python objects out of standard C++ variables and type.

The function is:

```
Py_BuildValue(char *format, ... )
```

It uses a format string for creating new Python objects out of the variable list given C++ variables.

This function is not yet started, but is needed in Pythons built-in types for building new objects in a standard and efficient way.
The syntax of the format string is as follows:

c char

A C++ char becomes a Python string with length one. It might be a QChar instead at a later time.

d double

A C++ double becomes a Python real.

D PY_Complex

A C++ complex becomes a Python complex.

i int

A C++ int becomes a Python int.

l long

A C++ long becomes a Python int.

S char* or QString later.

A C++ string becomes a Python string.

(...)

Build Python tuple from C++ values.

[...]

Build Python list from C++ values.

{ ... }

Build Python dictionary from c values. Alternating key and value as needed.

There will possible be more format characters later as needed, but this function is not yet started to be implemented.

For more details about the Python syntax for building values, you should look at Python in a nutshell page 524 - 525.

The error system

Python – Realizer (PR) needs a global error reporting system for executing of Python scripts and report errors as needed.

Pythons separate type library is supposed to be usable in plain C++ programs and also needs a way to signal error states in the objects.

I will try to use the C++ standard exception handling as a part of the total error handling in PR. This will take care of the needs of PR as well as stand alone C++ program using PR's type system. Also the C++ systems strong type system will be of great value in making PR a safer system than the original C based Python.

For the time being there is no error system defined for the data types of Python, when used as separate component to C++ programs.

I am thinking about a system where each objects knows its error state, and possibly can be interrogated for its error messages as needed.

Currently the parser reports errors it finds in the source code directly to the interpreter window for letting the user know about its failure.

I have started to think about a global error reporting system design, and will try to make it as similar to the standard C based Python as possible.

I will define standard error reporting methods for the system to be provided by the reporting method about what went wrong during the executing of the Python scripts.

I intend to follow the Pythons error systems defined error types, like value error, out of range error and have pre made methods or functions to be called when those states is found during executing of the Python scripts, and automatically generate needed error messages to the user through the interpreter window.

Currently there is not much of an error system defined or implemented. Some code has been implemented as part of the parsing of argument methods described earlier.

This needs to be addressed shortly, when more of the interpreter gets written.

Compiler system

With the compiler system, I mean the component that takes the parsers prepared abstract syntax tree with all the details stored in each node, and translates it to Python code objects with needed operands stored in arrays, and all the needed Python constants and local variables prepared in the frame objects.

I haven't started to think about how I will implement this component, except for two sure decisions already made. That is it must be 100% compatible with the standard Python instructions, and it must be able to create correct binary pre compiled files out of the Python source code.

This means that my compiler should be able to create the code objects which can run on standard Python and the other way around.

It is very important that the binary format of pre compiled source files is in exactly the same format, and follows the guidelines of Pythons marshal sub system.

The operands used in Python virtual machine is described a bit in the next chapter.

You will find more information about the compilers implementation in the C based python in the following two files:

> *compiler.h*
> *compiler.c*

Marshal system

This is the name of the format used on pre compiled files with the ending *.pyc* after its name instead of *.py* in the source files.

Pythons marshal system is implemented as a standard module and even has a few methods available.

Those are as follows:

```
dump
load
dumps
loads
```

It contains a predefined collection of API functions for marshal functionality in the interpreter and elsewhere if needed. They are as follows:

```
PyMarshal_WriteLongToFile
PyMarshal_WriteObjectToFile
PyMarshal_WriteObjectToString
PyMarshal_ReadLongFromFile
PyMarshal_ReadShortFromFile
PyMarshal_ReadObjectFromFile
PyMarshal_ReadLastObjectFromFile
PyMarshal_ReadObjectFromString
```

These are mostly for reading and writing of objects, integer and strings to and from the object files. It is a little like the C++ serializing functionality of class state to files for later recreating of class with the same state and data.

Marshal uses an object marker in a file to mark what kind of object follows in the file and thereby which method in the marshal class needs for reading it.

Those markers are as follows:

```
TYPE_NULL       '0'
TYPE_NONE       'N'
TYPE_FALSE      'F'
TYPE_TRUE       'T'
TYPE_STOPITEN   'S'
```

TYPE_ELLIPSIS	'.'
TYPE_INT	'i'
TYPE_INT64	'I'
TYPE_FLOAT	'f'
TYPE_COMPLEX	'x'
TYPE_LONG	'l'
TYPE_STRING	's'
TYPE_INTERNED	't'
TYPE_STRINGREF	'R'
TYPE_TUPLE	'('
TYPE_LIST	'['
TYPE_DICT	'{'
TYPE_CODE	'c'
TYPE_UNICODE	'u'
Might not be used.	
TYPE_UNKNOWN	'?'

I am not sure where to place the compiler and marshal functionality in the interpreter system of PR at this time. It might end up in the virtual machine library or as part of the application among the parser system.

Both systems will most likely be implemented as two separate classes with all needed functionality integrated for a separate component design.

Neither is started to be implemented or for that matter designed at this time, but will be the next logical step after the completion of parser.

All objects saved to files need to be edian[25] neutral for portability between the platforms with different edian format in the processor implementation.

More details about the standard C based Pythons marshal sub system is found in the following two files:

marshal.h
marshal.c

I have not started to implement the Python - Realizer (PR) edition of the marshal sub system at this time.

[25] Different processors have different format for the storing of data longer than 8 bits. Intel and Amd use little edian format while others mostly use big edian, or they can be programmed for both.

Virtual executing machine

I want my virtual execute machine for Python - Realizer to execute the pre compiled source code by the standard C based Python, to be loaded and executed without any need to recompile before use.

Python uses a so called *byte code*[26] for identifying each operand from a binary compiled file or straight from the compiler in interpreter mode.

It uses a stack for pushing and popping of pre defined Python data objects, which it then does its business on as pre programmed in the program.

Byte code is then executed on a virtual execute engine that interprets each operand and optional arguments as needed. This engine will be implemented entirely in C++.

Operand codes

I will use standard Python operand codes in my engine to let it be as compatible as possible.

Those operands of byte code are as following:

STOP_CODE	0
POP_TOP	1
ROT_TWO	2
ROT_THREE	3
DUP_TOP	4
ROT_FOUR	5
NOP	9
UNARY_POSITIVE	10
UNARY_NEGATIVE	11
UNARY_NOT	12
UNARY_CONVERT	13
UNARY_INVERT	15
LIST_APPEND	18
BINARY_POWER	19
BINARY_MULTIPLY	20
BINARY_DIVIDE	21
BINARY_MODULO	22
BINARY_ADD	23
BINARY_SUBTRACT	24
BINARY_SUBSCR	25
BINARY_FLOOR_DIVIDE	26
BINARY_TRUE_DIVIDE	27
INPLACE_FLOOR_DIVIDE	28
INPLACE_TRUE_DIVIDE	29
SLICE	30 - 33
STORE_SLICE	40 - 43
DELETE_SLICE	50 - 53
INPLACE_ADD	55
INPLACE_SUBTRACT	56
INPLACE_MULTIPLY	57
INPLACE_DIVIDE	58
INPLACE_MODULO	59
STORE_SUBSCR	60
DELETE_SUBSCR	61
BINARY_LSHIFT	62
BINARY_RSHIFT	63
BINARY_AND	64
BINARY_XOR	65
BINARY_OR	66
INPLACE_POWER	67
GET_ITER	68
PRINT_EXPR	70
PRINT_ITEM	71
PRINT_NEWLINE	72
PRINT_ITEM_TO	73
PRINT_NEWLINE_TO	74
INPLACE_LSHIFT	75
INPLACE_RSHIFT	76
INPLACE_AND	77
INPLACE_XOR	78
INPLACE_OR	79
BREAK_LOOP	80
LOAD_LOCALS	82
RETURN_VALUE	83
IMPORT_STAR	84
EXEC_STMT	85
YIELD_VALUE	86
POP_BLOCK	87
END_FINALLY	88
BUILD_CLASS	89

[26] This means op codes based on an 8 bits byte number to numerically describe an operation to be executed by the virtual machine.

HAVE_ARGUMENT	90 #Marker!
STORE_NAME	90
DELETE_NAME	91
UNPACK_SEQUENCE	92
FOR_ITER	93
STORE_ATTR	95
DELETE_ATTR	96
STORE_GLOBAL	97
DELETE_GLOBAL	98
DUP_TOPX	99
LOAD_CONST	100
LOAD_NAME	101
BUILD_TUPLE	102
BUILD_LIST	103
BUILD_MAP	104
LOAD_ATTR	105
COMPARE_OP	106
IMPORT_NAME	107
IMPORT_FROM	108
JUMP_FORWARD	110
JUMP_IF_FALSE	111
JUMP_IF_TRUE	112
JUMP_ABSOLUTE	113
LOAD_GLOBAL	116
CONTINUE_LOOP	119
SETUP_LOOP	120
SETUP_EXCEPT	121
SETUP_FINALLY	122
LOAD_FAST	124
STORE_FAST	125
DELETE_FAST	126
RAISE_VARARGS	130
CALL_FUNCTION	131
MAKE_FUNCTION	132
BUILD_SLICE	133
MAKE_CLOSURE	134
LOAD_CLOSURE	135
LOAD_DEREF	136
STORE_DEREF	137
CALL_FUNCTION_VAR	140
CALL_FUNCTION_KW	141
CALL_FUNCTION_VAR_KW	142
EXTENDED_ARG	143

In addition to these operands, we have some sub operands for the comparing instruction for defining condition for jump in code.

Those are symbol representation for the following compare operators:

<	<=	==	!=	>	>=
IN	NOT IN		IS	IS NOT	
MATCH		BAD			

These are defined with an enumeration type instead of the listed symbols.

Most of the operands in the table are self explained by its name. A few has more than one value for an operand.

All operands with value higher than or equal to 90, takes arguments in addition to just the operand code.

Most operands work with Python objects, already pushed on the executing stack, or operate on references to object stored in the symbol table for global access.

Constants are stored in the code object for easy access just through a simple index into the table. Local variables might be stored in the current frame object, which holds the execute state for current code. Is it in a function or is it running at top level and so one.

All op codes in use are defined in the following file in standard C based Python:

opcode.h

Implementation details

I have just started implementing this execute engine for Python binary code, and have concentrated on the operand decoding and executing of them first.

The whole virtual machine is implemented as a dynamically loadable library, which can be used by the Realizer application or inside of a C++ program if needed.

ceval.h
ceval.c

It will later have a symbol table system for the storage of symbols used in executing of the program and a much needed error reporting system.

I am considering moving the parser into this library for a complete Python interpreter sub system. For now the parser stays in the main application.

We will use a Python compatible marshalling system for reading in or writing out Python objects and codes into a binary format to be stored in files.

All codes to be executed are stored in a code object, where we use an instruction pointer to get the current operand code out of data storage in object. We will use C++ operators like [] to access this easily, and add some boundary check on this pointer to keep code in valid memory area of the object in case of malfunction in the compiler phase.

Such a code object will always be part of a frame object, which stores the current running status and context for the engine to work under.

I have only started to implement the operand decoding and executing of them at this time, and of course it is not ready to run programs for quite some time.

Details about the implementation of the standard C based Python is found in the following two files:

Extensions modules in C++

Most of Pythons library modules with useful functions to be reused as needed are written in Python, but quite a few of them are written in C for speed or functionality not possible directly in Python code.

You may also creates your own extensions module and add it to this module library.

All extension modules not written in Python, in the standard C based Python interpreter is of course written in C and not exactly what I want to use in a C++ interpreter.

I have therefore been thinking about writing a C++ edition of most of these modules and put them into yet another dynamical loadable library for easy access also from C++ programs.

I haven't started to design or implement this part of the system, and think it might be the last sub system to be implemented.

Since this is going to be a complete C++ implemented interpreter system with Unicode as standard for all the text handling, not all of the extension modules make sense to be ported.

Some of the extension modules are for interfacing with other GUI systems, like the Tkinter[27] system. We are going to be using the Qt framework exclusively for all the GUI work, and therefore we need to design the interface between Qt and Python and maybe simplify this to suite our needs for a fast and easy form designs.

Other GUI sub system is not really needed, and might be dropped all together from the module library.

I am not going to mention or describe all modules needed to be implemented as we are finishing the C++ interpreter edition of Python, but I will mention some of them for a little preview of the work needed.

We have modules for among other things, zip and bzip handling of text and binary data in files. We have a module for interfacing with the Unix operating systems *posix* sub system, which is Unix's interchangeable interface to operating system calls across all the dialects of Unix and also supported in Windows.

We have modules for cryptographic handling of files, handling of graphics files and support for the GL graphic library for 3D graphic.

We have modules for signal handling in C, thread management in own programs, and a very useful module for socket communication over networks or locally on a computer.

For a full overview of all the modules, you can download the Python source code from www.python.org and look into the directory "Modules" for a full listing of modules and all the details therein in the C source files or in the Python documentation found on the same site.

[27] This GUI system was designed for use by the TCL language, but is now used as well as the standard GUI in standard C edition of Python.

The road ahead

This year worth of work on my master thesis has just scratched the surface on this big project.

I have made a skeleton application with some of the needed functionality included, and a mock up of what I want the application to look like when eventually finished.

I have mostly finished the parser and have an interpreter component, which will check the users test scripts or directly input statements, for the correct grammar in accordance with the Python languages grammar rules.

It is also possible to create and edit python scripts in the editor system, but it still has its quirks.

I have started on Pythons built in type library, and coded some of the functionality into it. I also made a template for how all the data types should be integrated in a type system. Still, a lot is missing and I suggest that this is the first component to be finished in further development of this project.

This library is also possible to use as a standalone library for C++ programs that want to have some Python functionality easily built in.

The compiler component is not started and is together with the virtual executing engine, the next logical step to finish. I have a complete abstract syntax tree generated by the finished parser to start working with, and the finished product for the compilers part is the code object.

The virtual executing engine is started to be implemented and only the op code execute method is partly finished as a template for the complete runtime system. Subsystems like marshalling of the data and the code to and from library modules is not started. This should be developed and designed for 100% compatibility with the standard C based Python for easy exchange of pre compiled modules.

All of the C coded extensions modules is missing, and is planned to be recoded as a C++ library instead. I suggest this is done last, since it is not that important in the beginning.

All Python coded library modules is supposed to just recompile on Python – Realizer, and be useful without any modifications when the compiler is finished. Logically this is to be done after the compiler is finished.

Finally I think the rest of applications form design, and other functionality should be implemented. And of course we need to document it all in the end to make it useful for most users of the system.

This project should make the base for several full year master students until it is close to be finished, or it could become an open source project instead.

During this year long project, I made a few detours into dead end design decisions. The parser is for one thing, rewritten about ten times, before I got something that I found useful. Even the built in data types have been implemented in two different test system, and the first was not something I was happy with. A lot of time was wasted in the pursuit of a

custom widget for the interpreter written from the ground and up.

To bad one doesn't have the knowledge of the end of the year in the beginning of the project.
My goal for my master thesis was to make the foundation for the whole system with most of its sub systems started.

With this is mind I have tried to implement something in most subsystems to have that foundation.

I suggest that future master thesis students on this project should possibly try to concentrate on finishing one subsystem at the time instead of spreading the resources all over. I think the built in data type library, and the virtual machine with the rest of the compiler should take precedence over the application development and library modules in both C++ and Python.

Appendix A

Full grammar of Python 2.4

```
Grammar:

Single_input:      <NEWLINE> | simple_stmt | compound_stmt <NEWLINE>
File_input:        ( <NEWLINE> | stmt )* <EOF>
Eval_input:        testlist (<NEWLINE>)* <EOF>

Decorator:         '@' dotted_name [ '( arglist ')' ] <NEWLINE>
Decorators:        ( Decorator )+
Funcdef:           [ Decorators ] <DEF> <NAME> Parameters ':' Suite
Parameters:        '('  [ Vararglist ] ')'
Vararglist:        ( Fpdef [ '=' Test ',' ]  )* ( '*' <NAME> [ ',' '**' <NAME> ] |
                   '**' <NAME> ) | Fpdef [ '=' Test ] ( ',' Fpdef [ '=' Test ] )* [ ',' ]

Fpdef:             <NAME> | '(' Fplist ')'
Fplist:            Fpdef ( ',' Fpdef )* [ ',' ]

Stmt:              Simple_stmt | Compound_stmt
Simple_stmt:       Small_stmt ( ';' Small_stmt )* [ ';' ] <NEWLINE>
Small_stmt:        Expr_stmt | Print_stmt | Del_stmt | Pass_stmt | Flow_stmt |
                   Import_stmt | Global_stmt | Exec_stmt | Assert_stmt

Expr_stmt:         Testlist ( Augassign  Testlist |  ( '=' Testlist )* )
Augassign:         '+=' | '-=' | '*=' | '/=' | '%=' | '&=' | '|=' | '^=' | '<<=' |
                   '>>=' | '**=' | '//='
Print_stmt:        <PRINT> ( [ Test ( ',' Test )* [ ',' ] |
                   '>>' Test  [ ( ',' Test )+ [ ',' ] ] )
Del_stmt:          <DEL> Exprlist
Pass_stmt:         <PASS>
Flow_stmt:         Break_stmt | Continue_stmt | Return_stmt | Raise_stmt |
                   Yield_stmt
Break_stmt:        <BREAK>
Continue_stmt:     <CONTINUE>
Return_stmt:       <RETURN> [ Testlist ]
Yield_stmt:        <YIELD> Testlist
Raise_stmt:        <RAISE> [ Test [ ',' Test [ ',' Test ] ] ]
Import_stmt:       Import_name | Import_from
Import_name:       <IMPORT> Dotted_as_names
Import_from:       <FROM> Dotted_name <IMPORT> ( '*' | '(' Import_as_names ')'
                   | Import_as_names
Import_as_name:    <NAME> [ <NAME> <NAME> ]
Dotted_as_name:    Dotted_name [ <NAME> <NAME> ]
Import_as_names:   Import_as_name ( ',' Import_as_name )* [ ',' ]
```

```
Dotted_as_names:  Dotted_as_name ( ‘,’ Dotted_as_name )*
Dotted_name:      <NAME> ( <NAME> )*
Global_stmt:      <GLOBAL> <NAME> ( ‘,’ <NAME> )*
Exec_stmt:        <EXEC> Expr [ <IN> Test [ ‘,’ Test ] ]
Assert_stmt:      <ASSERT> Test [ ‘,’ Test ]

Compound_stmt:    If_stmt | While_stmt | For_stmt | Try_stmt | Funcdef |
                  Classdef
If_stmt:          <IF> Test ‘:’ Suite ( <ELIF> Test ‘:’ Suite )* [ <ELSE> ‘:’ Suite ]
While_stmt:       <WHILE> Test ‘:’ Suite [ <ELSE> ‘:’ Suite ]
For_stmt:         <FOR> Exprlist <IN> Testlist ‘:’ Suite [ <ELSE> ‘:’ Suite ]
Try_stmt:         ( <TRY> ‘:’ Suite ( Except_clause ‘:’ Suite )+ [ <ELSE> ‘:’ Suite ]
                  | <TRY> ‘:’ Suite <FINALLY> ‘:’ Suite )
Except_clause:    <EXCEPT> [ Test [ ‘,’ Test ] ]
Suite:            Simple_stmt | <NEWLINE> <INDENT> ( stmt )+ <DEDENT>

Test:             And_test ( <OR> And_test )* | Lambdef
And_test:         Not_test ( <AND> Not_test )*
Not_test:         <NOT> Not_test | Comparison
Comparison:       Expr ( Comp_op Expr )*
Comp_op:          ‘<’ | ‘>’ | ‘==’ | ‘>=’ | ‘<=’ | ‘<>’ | ‘!=’ | <IN> | <NOT> <IN> |
                  <IS> | <IS> <NOT>
Expr:             Xor_expr ( ‘|’ Xor_expr )*
Xor_expr:         And_expr ( ‘^’ And_expr )*
And_expr:         Shift_expr ( ‘&’ Shift_expr )*
Shift_expr:       Arith_expr ( ( ‘<<’ | ‘>>’ ) Arith_expr )*
Arith_expr:       Term ( ( ‘+’ | ‘-‘ ) Term )*
Term:             Factor (  ( ‘*’ | ‘/’ | ‘%’ | ‘//’ ) Factor )*
Factor:           ( ‘+’ | ‘-‘ | ‘~’ ) Factor | Power
Power:            Atom ( Trailer )* [ ‘**’ Factor ]
Atom:             ‘(‘ [ Testlist_gexp ] ‘)’ | ‘[‘ [ Listmaker ] ‘]’ |
                  ‘{‘ [ Dictmaker ] ‘} | ‘`’ Testlist1 ‘`’ | <NAME> | <NUMBER> |
                  ( <STRING> )+

Listmaker:        Test ( List_for | ( ‘,’ Test )* [ ‘,’ ] )
Testlist_gexp:    Test ( Gen_for | ( ‘,’ Test )* [ ‘,’ ] )
Lambdef:          <LAMBDA> [ Varargslist ] ‘:’ Test
Trailer:          ‘(‘ Arglist ‘)’ | ‘[‘ Subscriptlist ‘]’ | ‘.’ <NAME>
Subscriptlist:    Subscript ( ‘,’ Subscript )* [ ‘,’ ]
Subscript:        ‘.’ ‘.’ ‘.’ | Test | [ Test ] ‘:’ [ Test ] [ Sliceop ]
Sliceop:          ‘:’ [ Test ]
Exprlist:         Expr ( ‘,’ Expr )* [ ‘,’ ]
Testlist:         Test ( ‘,’ Test )* [ ‘,’ ]
Testlist_safe:    Test [ ( ‘,’ Test )+ [ ‘,’ ] ]
Dictmaker:        Test ‘:’ Test ( ‘,’ Test ‘:’ Test )* [ ‘,’ ]

Classdef:         <CLASS>  <NAME> [ ‘( Testlist ‘)’ ] ‘:’ Suite
```

```
Arglist:        ( Argument ',' )* ( Argument ','  | '*' Test [ ',' '**' Test ]
                | '**' Test )
Argument:       [ Test '=' ] Test [ Gen_for ]

List_iter:      List_for | List_if
List_for:       <FOR> Exprlist <IN>  Testlist_safe [ List_iter ]
List_if:        <IF> Test [ List_iter ]

Gen_iter:       Gen_for | Gen_if
Gen_for:        <FOR> Exprlist <IN> Test [ Gen_iter ]
Gen_if:         <IF> Test [ Gen_iter ]

Testlist1:      Test ( ',' Test )*
```

Notation for the grammar language:

< … > is Token, ( ) is a grammar rule grouping, [ ] is the optional grammar rule.
* means none to many, + means one to many, ' .. ' means the character between quotes.
…. : is the name of grammar rule.

Appendix B.

Building of the application and install Qt.

You will need to download, build and install the Qt library from Trolltech, if your computer doesn't have it already installed as most Linux distribution does.

My project have been tested against Qt version 3.2.1 and 3.3.4 and found to work with both.

It has been build and tested on Windows XP, Linux Fedora core 3 and Macintosh OSX 10.3 and found to work.

If you don't have Qt installed with thread support, you must download Qt source code from www.trolltech.com and build it yourself.

Instructions for building Qt:

```
Download archive file:

        qt-x11-free-3.3.4.tar.gz

Unpack it to your home path:

tar -xvzf qt-x11-free-3.3.4.tar.gz

cd qt-3.3.4

export QTDIR=$PWD

./configure -thread -fast -shared

make

make install  ( as root )
```

You will of course need a C++ compiler suite for your platform to let Qt be build and installed at all.
To build my application, download the newest archive from www.python-realizer.net and select the download menu at the index page to get the newest archive of the source code.

Look for something like:

realizer_mmm_dd_2005.tar.gz

Where mmm is month and dd is day.
Download the latest edition and follow the build instructions below:

```
tar -xvzf realizer_xxx_dd_2005.tar.gz

cd kildekode

qmake

make

On a Linux machine you will start
application by:

Cd Realizer

./Realizer
```

You will find the dynamic loadable library in the same directory.

To check the status of all the source files in project, and you will need a Python interpreter installed on your machine to run it.

Just type ./status.py

Literature reference

All the Python source code was found at www.python.org and I used the version 2.4.0 as a reference system.

All the documentation for Qt classes and methods was found in the document in html format included with source code downloaded from Trolltech's web at www.trolltech.com

Books used under development of project:

Python in a nutshell -
Covers Python 2.2
O'Reilly & Associates, 2003

C++ in a nutshell
Covers ISO/IEC 14882 Std.
O'Reilly & Associates, 2003

C++ GUI programming with Qt 3.
By Jasmin Blanchette and Mark Summerfiled.
Prentice Hall, 2004

Programming with Qt, 2nd edition.
Matthias Kalle Dalheimer.
O'Reilly & Associates, 2002

Datateknologiens utvikling.
Per A. Holst.
Tapir Akademisk Forlag

The C++ Programming language.
3. Edition
Bjarne Stroustrup
Addison - Wesley, 1997

Colophon

This master thesis report was written on a Portable Windows XP based computer with Office 2003 as the word processor.

I use the font "Trebucket" for all my writing, and the harmony template for document writing.

All images of screen dump was grabbed on a dual Power Macintosh machine with Mac OSX 10.3.9 and the utility "grab" which takes snapshots of the screen.

Final conversion to Acrobat reader format was done by Microsoft Office for Macintosh's word 2004.

All development of the software for use in this project was done on the same Macintosh, with GNU software's C++ compiler version 3.4.3 and utilities.

It was also tested on the Qt 3.2.3 under Windows XP with the Borland C++ compiler suite.