MINISTRY OF EDUCATION AND SCIENCE OF REPUBLIC OF KAZAKHSTAN
SULEYMAN DEMIREL UNIVERSITY
ENGINEERING FACULTY

Department of Computer Engineering

UDC 004.420

Kopbayeva Zhanat

ANALYSIS OF SYSTEMS PYTHON VS. RUBY

6M070400– «Computing systems and software» speciality

Kaskelen, 2013

MINISTRY OF EDUCATION AND SCIENCE OF REPUBLIC OF KAZAKHSTAN
SULEYMAN DEMIREL UNIVERSITY
ENGINEERING FACULTY

Department of Computer Engineering

«Accepted for defense»
responsible director of department
d.t.s professor Amirgaliyev E. N.
_____
«__» _____ 2013 year

Director of department
for postgraduate education
Ph.s.c. Aydogmush Sh.
«___» _____ 2013 year

Master dissertation

ANALYSIS OF SYSTEMS PYTHON VS. RUBY

6M070400– «Computing systems and software» speciality

Master student _____ Kopbayeva Zhanat
(signature)

Supervisor
_____ d.t.s., professor Niyazi Ari
(signature)

Kaskelen, 2013

CONTENTS

NORMATIVE REFERENCES

TIOBE index – The coding standards company. Here the programming languages are rated due to different criteria using popular search engines [19]:

- Google: 30%
- Blogger: 30%
- Wikipedia: 15%
- YouTube: 9%
- Baidu: 6%
- Yahoo!: 3%
- Bing: 3%
- Amazon: 3%

ISO/IEC 30170 - is the international standard for Ruby language. It describes the Ruby syntax, the execution of Ruby code, and a small library of classes and modules, with enough detail for 313 pages. The mruby implementation targets this standard. ISO/IEC 30170 omits part of Ruby's core library, and is not enough to run some Ruby programs. [18]

JIS X 3017:2011 – Ruby code standards from Japanese Institution of Standardization.

DEFINITIONS

1. YARV - (Yet another Ruby VM) is a bytecode interpreter that was developed for the Ruby programming language by Koichi Sasada. The goal of the project was to greatly reduce the execution time of Ruby programs.Since YARV has become the official Ruby interpreter for Ruby 1.9, it is also named **KRI**(Koichi's Ruby Interpreter).

2. Scripting language - a programming language that supports the writing of **scripts**, programs written for a special runtime environment that can interpret and automate the execution of tasks which could alternatively be executed one-by-one by a human operator.

3. System programming language - a language designed for writing system software as distinct from application software.

4. Compiler - a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program.

5. Interpreter - a computer program that executes, i.e. *performs*, instructions written in a programming language. An interpreter generally uses one of the following strategies for program execution:

1.execute the source code directly

2.translate source code into some efficient intermediate representation and immediately execute this

3.explicitly execute stored precompiled code[1] made by a compiler which is part of the interpreter system

6. POLA - principle of least astonishment.

7. Encoding Standard - An **encoding** defines a mapping from a code point sequence to a byte sequence (and vice versa).

8. Multi-platform - If a software program is developed for mulitple operating systems, it is considered to be "multiplatform."

9. GUI – Graphical User Interface

10. OOP – Object Oriented Programming

11. Class - In object-oriented programming, a **class** is a construct that is used to create instances of itself – referred to as *class instances*, *class objects*, *instance objects* or simply *objects*. A class defines constituent members which enable its instances to have state and behavior.

12. Subclass - A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*).

13. Inheritance - classes can be *derived* from other classes, thereby *inheriting* fields and methods from those classes.

14. Source lines of code (SLOC) - a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code.

15. Source code - any collection of computer instructions(possibly with comments) written using some human-readable computer language, usually as text. The source code of a program is specially designed to facilitate the work of computer programmers, who specify the actions to be performed by a computer mostly by writing source code. The source code is often transformed by a compiler program into low-level machine code understood by the computer. The machine code might then be stored for execution at a later time. Alternatively, an interpreter can be used to analyze and perform the effects of the source code program directly on the fly.

16. Iterator - any object that, pointing to some element in a range of elements (such as an array or a container), has the ability to iterate through the elements of that range using a set of operators.

17. Multiple Inheritance - a feature of some object-oriented computer programming languages in which a class can inherit characteristics and features from more than one superclass. It is distinct from single inheritance, where a class may only inherit from one particular superclass.

18. Super class - The class from which the subclass is derived is called a *superclass* (also a *base class*or a *parent class*).

Abstract

Nowadays web development programming is very popular. Using object-oriented programming (OOP) languages in it made it fun to design overall architectures, functionality and ease of usability. As the scripting languages did not lose their popularity in this process, on their own field they also were making progress for making web development interesting for a programmer. Also scripting languages' popularity is stable because of their compatibility and ease of use with the other languages. Considering these in this paper the comparison of basic items is made, that exist in Ruby and Python.

Python itself was already popular, but coming of Ruby into the world of developers made them to begin many discussions. This work will give you basic view to make yourself a conclusion which one to choose and here it's assumed that you already know one of the OOP languages or at least have some basic understanding of it.

It won't be just the basic interview with Python and Ruby, but from the introduction part the main concepts for language differences like dynamic versus static typing, strong and weak typing, compiled versus interpreted properties' overview will be made.

At the end developers will be able to conclude for themselves how to start and continue, while making clear reasoning. Because the work was made to do a rational overview of both Ruby and Python so that there won't be any prejudice of a particular individual.

Абстракт

В современном мире веб программирование очень популярно. Использование в них объектно-ориентированных языков программирования (ООП) сделало интересным процесс дизайна всей архитектуры, функционалов и легкости употребления. И в этом процессе скриптовые языки тоже не теряли свою популярность, в своей области они тоже прогрессировали, чтобы для разработчика было интересно веб программирование в целом. К тому же их популярность стабильно, потому что они совместимы и легко используются наряду с другими языками. Учитывая это, в этой работе было приведено сравнение основных элементов, которые есть в языках Руби и Питон.

Питон сам по себе был уже популярен, но приход Руби в мир разработчиков заставило их начать множество дискуссии. Эта работа даст вам основной обзор, чтобы вы могли сделать для себя вывод, который из них выбирать и здесь учитывается, что вы уже знаете одну из языков ООП или минимум имеете некоторые основные понятия концепции ООП.

Это не будет только знакомство с Питон и Руби в общем, но из введения основные понятия для различии языков как динамическое и статистическое программирование, строгий и слабый контроль типов, компилированные и интерпретированные свойства будут рассмотрены.

В конце разработчики смогут сделать вывод, как начинать и продолжить при этом основываясь на четко поставленных причинах. Потому что работа была сделана для рационального обзора обеих языков, чтобы не было никаких предубеждении.

Түйін

Қазіргі таңда интернеттің дамуына байланысты веб интерфейстің бағдарламалық қамтамасыз етілуі де өз өзектілігін сақтауда. Оның ішінде объектілі-бағытталған бағдарламалау тілдерін (ОББ) қолдану жалпы құрылым, функционалдар мен қолданыс ыңғайлылығын рәсімдеу процесін қызықты етті. Және бұл процессте скриптік тілдер өз өзектілігін жоғалтпады, өз алаңында олар да прогресспен алға басты және сол арқылы бағдарламаушыларға веб бағдарламалауды қызықты етті. Оған қоса олардың өзектілігі тұрақты, өйткені олар басқа бағдарламалау тілдерімен үйлесімді және қатар қолданғанға ыңғайлы. Осыны ескере отырып, осы жұмыста Руби және Питон тілдеріндегі негізгі элементтердің салыстыруы жасалынды.

Питон одан алдын да танымал болатын, ал Рубидің бағдарламаушылардың әлеміне кіруі олардың көптеген пікірталастарына себеп болды. Бұл жұмыс сіздерге жалпы таныту жасайды, сол арқылы сіздер екеуінің қайсысын таңдайтыныңызға шешім жасай аласыздар және бұл жерде сіз ОББ тілдерінің бірін біледі немесе ОББ концептілерінің негізгі түсінігіне ие деп қабылдануда.

Бұл тек қана Руби және Питонмен жалпы танысу ғана болмайды, бірақ кіріспеден тілдердің өзгешелігіне байланысты негізгі мынадай түсініктер қарастырылады: динамикалық және статистикалық бағдарламалау, қатал және әлсіз тип бақылауы, компиляцияланған және интерпретацияланған тілдер қасиеті.

Соңында бағдарламалаушылар өздеріне қалай бастау және жалғастырып алып кететіндігіне шешімді нақты себептерге негізделе отырып қабылдай алады. Өйткені жұмыс Питон және Руби тілдерінің рационалды түрде қарастыру үшін жасалынды, осы арқылы басқа пікірлерді араластырмау көзделді.

1. Introduction

1.1. Problems of choosing a language

For a developer used to work with system languages like C++ or Java a question like "why do I need to know a script language?" may be common. So let's discuss!

In coding with system languages the compiled code is used and with low performance computers it's good to use (Nowadays our computers work really fast so there's no need for a developer to distinguish between these two types according to the performance). While in scripting its code doesn't need to be compiled because it's already interpreted.

There are two important tradeoffs: interpreted vs. compiled and static vs. dynamic typing.

- Interpreter: Fast to develop (edit and run). Slow to execute because each statement had to be interpreted into machine code every time it was executed (think of what this meant for a loop executed thousands of times).

- Compiler: Slow to develop (edit, compile, link and run. The compile/link steps could take serious time). Fast to execute. The whole program was already in native machine code.

A one or two order of magnitude difference in the runtime performance existed between an interpreted program and a compiled program. Other distinguishing points, run-time mutability of the code for example, were also of some interest but the major distinction revolved around the run-time performance issues.

Today the landscape has evolved to such an extent that the compiled/interpreted distinction is pretty much irrelevant. Many compiled languages call upon run-time services that are not completely machine code based. Also, most interpreted languages are "compiled" into byte-code before execution. Byte-code interpreters can be very efficient and rival some compiler generated code from an execution speed point of view.

The classic difference is that compilers generated native machine code, interpreters read source code and generated machine code on the fly using some sort of run-time system. Today there are very few classic interpreters left - almost all of

them compile into byte-code (or some other semi-compiled state) which then runs on a virtual "machine".

Python and Ruby, which we discuss about in this work are the ones compiling into byte-code.

Scripting languages are higher-level than system programming languages, as a single instruction does a lot more work, mostly due to the fact that primitive operations in scripting languages are far more powerful.[1]

As for the static vs. dynamic typing there's also consideration of strong and weak typed language features.

Static typed programming languages are those in which variables need not be defined before they're used.

Dynamic typed programming languages are those languages in which variables must necessarily be defined before they are used.

Programming languages in which variables have specific data types are strong typed. This implies that in strong typed languages, variables are necessarily bound to a particular data type.[11] Let's see this with an example of python code:

Python code:

```
>>> foo = "x"
>>> foo = foo + 2
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in ?
    foo = foo + 2
TypeError: cannot concatenate 'str' and 'int' objects
```

As the error shows the type of string cannot be used along with the int type. Thus this implies that Python is a strong typed language. So is the Ruby.

As opposed to strong typed languages, weak typed languages are those in which variables are not of a specific data type. It should be noted that this does not imply that variables do not have types; it does mean that variables are not "bound" to a specific data type. Examples of this types are Php, C.

Thus the scripting languages are typeless, meaning that they're dynamic typed, it makes them more flexible because the set of rules like in system language aren't much to be considered. Also it's easy to code and easy to use with other system languages with available libraries of system languages to embed the scripts.

System programming languages were designed for building data structures and algorithms from scratch, starting from the most primitive computer elements. Scripting languages are designed for gluing. They assume the existence of a set of powerful components and are intended primarily for connecting components. [2]

1.2. Advantages of using Python

Guido van Rossum is the author of Python, an interpreted, interactive object-oriented programming language. In the late 1980s, Van Rossum began work on Python at the National Research Institute for Mathematics and Computer Science in the Netherlands, or *Centrum voor Wiskunde en Informatica* (CWI) as it is known in Dutch. Since then, Python has become very popular among developers, who are attracted to its clean syntax and reputation for productivity. [3]

Its implementation was started in December 1989 by Guido van Rossum at CWI in the Netherlands as a successor to the ABC programming language capable of exception handling and interfacing with the Amoeba operating system. Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, *Benevolent Dictator for Life* (BDFL).

Python 2.0 was released on 16 October 2000, with many major new features including a full garbage collector and support for unicode. The last version released on September 29, 2012 is Python 3.3.

Python has been awarded a TIOBE Programming Language of the Year award twice (in 2007 and 2010), which is given to the language with the greatest growth in popularity over the course of a year, as measured by the TIOBE index. [4]

Table 1.2.1. – Applications with Python scripting [5]

Applications and Toolkits

Name	Platform	Notes
Abaqus	**multi-platform**	**Finite element analysis, modeling, and visualization. Python is embedded for command scripting and GUI extensibility.**
BioNumerics	**Windows**	**A suite of bioinformatics software applications,** uses Python **to automate series of actions that are executed repeatedly, to create custom reports, import and export non-standard formats, perform custom calculations, etc.**
ClearSilver	**multi-platform**	**Web application framework**
CodeSubWars	**Windows**	**Free programming game. Uses Python as scripting language for submarine robots.**
Emacs, using Pymacs	**multi-platform**	**Text editor.**

FontLab	**Windows, Mac OS**	**Commercial font editing software, uses Python as a macro language.**
GXSM	**Linux, Mac OS X**	**Gnome X Scanning Microscopy - multi channel 2D/3D data acquisition and visualization**
Helix DNA	**multi-platform**	**The Helix community is a collaborative effort among real, independent developers, and leading companies to extend the Helix DNA™ platform, the first open multi-format platform for digital media creation, delivery and playback.**
Kahakai Window Manager	**POSIX and X11**	**Python is used for event and key binding configuration.**
Kaydara Motionbuilder	**Mac OS X, Windows**	**Helpful for integrating into production pipelines and allows users to automate repetitive processes.**
Mahogany	**Linux, Mac OS X, Windows**	OpenSource **cross-platform mail and news client.**
GarageCube Modul8	**Mac OS X**	**Video mixing and compositing for VJs and live performers; uses Python for internal scripting.**
NSIS2	**Windows**	**There is an** NSIS Python Plugin **for this** OpenSource **installer for Windows. -- Note:***makensis*, **the installer compiler now compiles and runs on POSIX**

		systems too.
OpenOffice	multi-platform	Python-UNO bridge for scripting OpenOffice with Python
Office and Outlook	Windows	While Microsoft Office apps including Microsoft Outlook were not originally designed to be scripted with Python, PyWin32 (formerly Win32All) makes it possible to script Microsoft Office apps and build plug-ins. Of special note is the SpamBayes plug-in for Outlook which enables any Outlook user to use SpamBayes without even knowing anything about Python.
Orange	multi-platform	Orange is a component-based data mining software. It includes a range of preprocessing, modelling and data exploration techniques. It is based on C++ components, that are accessed either directly (not very common), through Python scripts (easier and better), or through GUI objects called Orange Widgets.
Orca	GNOME	Accessibility toolkit for applications using GTk+/AT-SPI that provides speech synthesis, braille, and magnification.
Paint Shop Pro 8	Windows	Photo and graphics editor. PSP scriptin resources

Panda3D	**multi-platform**	**Open Source game and simulation engine. Disney is one of the notable users.**
ProScena **Studio**		**A high-level content-authoring system and runtime for creating real-time interactive 3D simulations for training, education and gaming applications.**
Quantum GIS	**multi-platform**	**User friendly open source Geographic Information System (GIS)**
Resolver One	**.NET/CLR**	**Programmable spreadsheet built around** IronPython.
Scorpion Vision Software	**Windows**	**Machine-vision software for industrial use with Python scripting**
SilverRun ModelSphere	**Windows/Linux/Solaris**	**A relational data modeling tool with Jython scripting.**
Squish	**multi-platform**	**An automated GUI testing framework for Qt applications that supports Python scripting**
Subversion	**multi-platform**	**A compelling replacement for CVS. Se the** Project Links, Bindings **or do a** Google site search.
SuperKaramba	**Linux, KDE**	SuperKaramba **allows you to create cool dekstop widgets with little to no programming experience. It's similar t**

		"Konfabulator" for the Mac.
truSpace	**Windows**	**3D modelling/animation package**
Vim	**multi-platform**	**Text editor.**
WinCvs	**Windows**	**GUI front-end for CVS**
XChat , SilvereX's free Windows version	**multi-platform**	**IRC Client**

Graphics

Acorn	MacOS X	Image editor.
Blender	multi-platform	OpenSource software for 3D modeling, animation, rendering, post-production, interactive creation and playback. Blender Python API reference and tutorials.
Crystal Space	multi-platform	OpenSource 3D SDK
GIMP	multi-platform, GTK	GNU Image Manipulation Program. PyGimp is a package that allows people to write plug-ins for Gimp on Python rather than Script-Fu (Scheme), Perl, Tcl or C.
Inkscape	multi-platform	Scalable Vector Graphics editor

irrlicht	multi-platform	The Irrlicht Engine is an open source high performance realtime 3D engine written and usable in C++ and also available for .NET languages. It is completely cross-platform, using D3D, OpenGL and its own software renderer, and has all of the state-of-the-art features which can be found in commercial 3d engines.

1.3. Advantages of using Ruby

Ruby was first designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.

The standard 1.8.7 implementation is written in C, as a single-pass interpreted language. Starting with the 1.9 branch, and continuing with the current 2.0 branch, YARV has been used, and will eventually supersede the slower Ruby MRI. The language specifications for Ruby were developed by the Open Standards Promotion Center of the Information-Technology Promotion Agency (a Japanese government agency) for submission to the Japanese Industrial Standards Committee and then to the International Organization for Standardization. It was accepted as a Japanese Industrial Standard (JIS X 3017) in 2011 and an international standard (ISO/IEC 30170) in 2012. [6]

Matsumoto has said that Ruby is designed for programmer productivity and fun, following the principles of good user interface design. He stresses that systems design needs to emphasize human, rather than computer, needs:[7]

- "Often people, especially computer engineers, focus on the machines. They think, "By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something." They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves."

Ruby is said to follow the principle of least astonishment (POLA). Matsumoto defined it this way in an interview: [7]

-"Everyone has an individual background. Someone may come from Python, someone else may come from Perl, and they may be surprised by different aspects of the language. Then they come up to me and say, 'I was surprised by this feature of the language, so Ruby violates the principle of least surprise.' Wait. Wait. The principle of least surprise is not for you only. The principle of least surprise means principle of least *my* surprise. And it means the principle of least surprise after you learn Ruby very well. For example, I was a C++programmer before I started designing Ruby. I programmed in C++ exclusively for two or three years. And after two years of C++ programming, it still surprises me."

Table 1.3.1. - Popular sites using Ruby

- Ask.fm
- Bleacherreport.com
- Intuit.com
- Patch.com
- Opensiteexplorer.org
- Pixlr.com
- Myfitnesspal.com
- Fab.com
- Dribbble.com
- Reverbnation.com

2. Differences between Ruby and Python.

2.1. Object Oriented Programming

It's true that both languages are OOP and this comes to be their similarities. But the difference comes when the logic of OOP usage starts. That is in Python it comes from defining new classes using the basic logic of programming and in Ruby everything comes to be object.

Both take diametrically opposite views in bringing the object-oriented paradigm to the procedural world of scripting. Python starts out as a procedural language and adds object-oriented features for a programmer who wants to use them, while Ruby starts out as a pure object-oriented language and then makes the code look like procedural program, thus making it accessible to the general user. [1]

Due to this difference, we have uniform access in Ruby, which we can't have in Python. e.g. to get the absolute value of -3 you would say -3.abs in Ruby, while you would do abs(-3) in Python.

2.2. Version Compatibility

Python 3.0 (also called "Python 3000" or "Py3K") was designed to rectify certain fundamental design flaws in the language (the changes required could not be implemented while retaining full backwards compatibility with the 2.x series, which necessitated a new major version number). The guiding principle of Python 3 was: "reduce feature duplication by removing old ways of doing things". [10]

Because of this policy, things available in Python2.x weren't available in Python3.x.

In Ruby features available in old versions are also supported in new versions.

2.3. Source File's encoding

For editor interfaces in Ruby there's a wide range of encoding functionality support while in Python it's only setting the recognizing of code lines' encoding as

defined in editor interface. That is to say the Ruby library support of various functionality is bigger than in Python. Following examples demonstrate them.

In Ruby it's available to encode the String from one coding system to another if the program does need it and this is called transcoding.

Table 2.3.1. Transcoding example of Ruby

Code lines:	Output:

# encoding: utf-8 	
ole_in_utf = "olé"
ole_in_utf.encoding
ole_in_utf.bytes.to_a
ole_in_8859 = ole_in_utf.encode("iso-8859-1")
ole_in_8859.encoding
ole_in_8859.bytes.to_a
	# => #<Encoding:UTF-8>
	# => [111, 108, 195, 169]
	# => #<Encoding:ISO-8859-1>
	# => [111, 108, 233]

Ruby I/O objects also support encoding and transcoding options and methods for using when reading, writing or while coding.

Here's an example used for file recognizing in iso-8859-1 coding and output writing in utf-8 coding:

Table 2.3.2. Transcode of data read and write example

Code lines:
f = File.open("iso-8859-1.txt", "r:iso-8859-1:utf-8") puts f.external_encoding.name line = f.gets puts line.encoding puts line
produces:
ISO-8859-1 UTF-8 i>?olAc

2.4. Indentation and *end* keyword

In Python there's no need for additional keywords like *begin* and *end* to recognize blocks, you just have to write with indentation to outline the block's contents. [9]

In Ruby it's needed to put keyword *end* or if without it then parentheses like (), {} are put to recognize blocks. Also in a method's call the parentheses are optional.

Table 2.4.1. – Example of optional parentheses in method call of Ruby.

Code lines:
def say_goodnight(name) result = "Good night, " + name return result end # Time for sleeping... puts say_goodnight("JohnBoy") puts say_goodnight "MaryEllen"
produces:
Good night, JohnBoy Good night, MaryEllen

2.5. The Logic of "Range"

The logic of range is covered in both as it must come to be similarity. Again as mentioned above Ruby's approach of objects made this logic more powerful, because Ruby has it as an object class "Range" and in Python it's used as a function "range()". Based on these Python's range iterates only over a sequence of numbers while Ruby's Range may iterate also using characters as well as class objects.

Table 2.5.1. – Example for Ruby's Range class

Code lines:

```
class PowerOfTwo
attr_reader :value
def initialize(value)
@value = value
end
def <=>(other)
@value <=> other.value
end
def succ
PowerOfTwo.new(@value + @value)
end
def to_s
@value.to_s
end
end
p1 = PowerOfTwo.new(4)
p2 = PowerOfTwo.new(32)
puts (p1..p2).to_a
```

produces:
4 8 16 32

2.6. The availability of different syntaxes.

Let's start the comparison of one coding example of both languages firstly looking in Python.

Table 2.6.1. – Python example of iteration.

Code lines:
for i in range(1,11): if(i % 2 == 0 and i > 5): print i,"\n"

And let's give the same code in different styles of Ruby coding example.

Table 2.6.2. – Examples of Ruby: short form.

Code lines:

```
(1..10).each { |i| print "#{i}\n" if i % 2 == 0 && i > 5 }
```

Table 2.6.3. – Examples of Ruby: more readable

Code lines:

```
(1..10).each do |i|
        if i % 2 == 0 and i > 5
            print i, "\n"
        end
end
```

Table 2.6.4. – Examples of Ruby: more readable without keyword *each*.

Code lines:

```
for i in 1..10 do
     if i % 2 == 0 and i > 5 then
              print i, "\n"
       end
end
```

As we can see from examples the Python code is more readable and understandable. Thus it comes to be that Python code is easier to learn.

It's true that Ruby developing team accomplished to give for different background programmers coming from different programming languages to have in Ruby their own coding style without any limitations. And this feature of Ruby comes to be confusing for a starting learner without much experience. So makes it difficult to learn Ruby at the beginning.

2.7. All constructs have a value.

A feature of Ruby distinction from Python is that all constructs in Ruby have a return value. Let's demonstrate this in an example.

Table 2.7.1. – Example of *if*-construct returning a value in Ruby.

Code lines:

```
txt = "Hello World"
a = "size " +
    if txt.size > 3 then
        "greater"
    else
        "less"
    end +
    " than 3"
print a
```

produces:
size greater than 3

2.8. Inheritance and Visibility.

In Ruby there's a wrinkle to when it comes to method definition and class inheritance, but it's fairly obscure. Within a class definition, you can change the visibility of a method in an ancestor class. [8]

Table 2.8.1. – Example of method's visibility changing from derived class in Ruby.

Code lines:
class Base def a_method puts "Got here" end private :a_method end class Derived1 < Base public :a_method end

```
class Derived2 < Base

end
```

In the example above we can invoke a_method from class Derived1 but not from class Derived_2.

If a subclass changes the visibility of a method in a parent, Ruby effectively inserts a hidden proxy method in the subclass that invokes the original method using super. It then sets the visibility of that proxy to whatever you requested. This means that the following code:

```
class Derived1 < Base

    public :a_method

end
```

is effectively the same as this:

```
class Derived1 < Base

    def a_method(*)

        super

    end

    public :a_method

end
```

The call to super can access the parent's method regardless of its visibility, so the rewrite allows the subclass to override its parent's visibility rules.

2.9. Multiple Inheritance.

Ruby allows only single inheritance, that is to avoid complexity problems that may come from multiple inheritance. Instead of it Ruby uses concept of modules. When a module is added to the class: the module that you include becomes a superclass of the class being defined and the instance methods in that module become available as instance methods of the class. [8]

Python allows multiple inheritance. A classic issue with multiple inheritance is name conflicts between super-classes. Python "solves" this simply by having the programmer annotate the order of superclasses, and then search class by class for the name until it is found by depth-first and from left-to-right. [1]

Table 2.9.1. – Module using syntax in Ruby example.

Code lines:

```
module Logger
  def log(msg)
    STDERR.puts Time.now.strftime("%H:%M:%S: ") + "#{self} (#{msg})"
  end
end
class Song
  include Logger
end
class Album
  include Logger
```

```
end
s = Song.new
s.log("created")
```

produces:

```
13:26:13: #<Song:0x0a323c> (created)
```

Table 2.9.2. – Multiple Inheritance syntax in Python. [9]

Code lines:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Python's multiple inheritance approach may lead to confusions even the experienced programmer when making big projects. While in Ruby's syntax this approach comes to be simpler to understand and neat to make further development.

3. SIMILARITY COMPARISON

To analyze more clearly the available functionality and performance of both languages let's discuss it with writing same logic of codes on same machines in both languages.

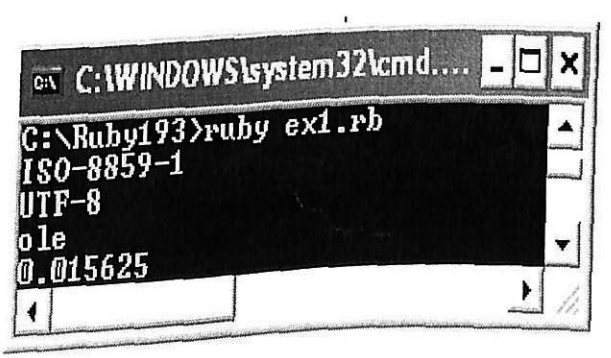
3.1. Source File's Encoding

Ruby example of transcoding of I/O objects:

Table 3.1.1. - Ruby Transcoding

Code line
t1=Time.now() f = File.open("iso-8859-1.txt", "r:iso-8859-1:utf-8") puts f.external_encoding.name line = f.gets puts line.encoding puts line puts Time.now()-t1

Figure 3.1.1. – Output of Ruby Transcoding example



Python example:

Table 3.1.2. - Python3.0 Transcoding

Code line
from time import time import codecs t1=time() file_stream = codecs.open("iso-8859-1.txt", 'r', 'iso-8859-1') l=file_stream.readline() print(l.encode('utf-8')) file_stream.close() print(time()-t1)

Figure 3.1.2. – Output of Python 3.0 Transcoding example

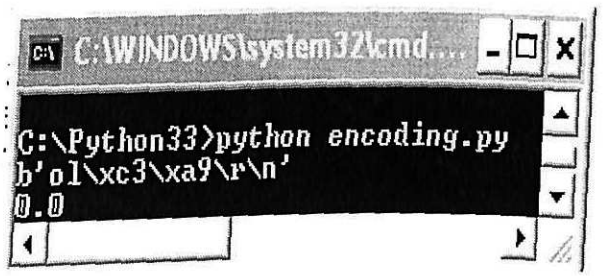


Table 3.1.3. - Python2.0 Transcoding

Code line

```
from time import time
import codecs
t1=time()
file_stream = codecs.open("iso-8859-1.txt", 'r', 'iso-8859-1')
l=file_stream.readline()
   print l.encode('utf-8')
file_stream.close()
print time()-t1
```

Figure 3.1.3. – Output of Python 2.0 Transcoding example

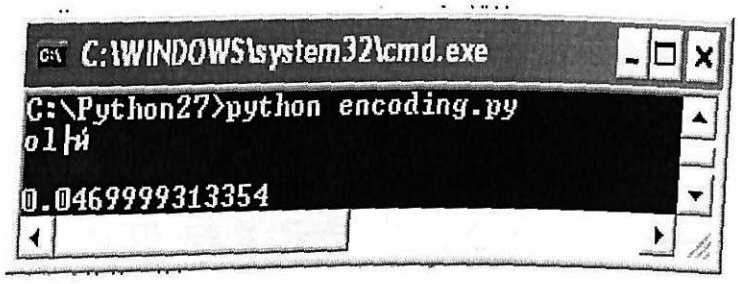


Table 3.1.4. - Python2.0 Transcoding example with writing output to a file.

Code line
from time import time
import codecs
t1=time()

```
file_stream = codecs.open("iso-8859-1.txt", 'r', 'iso-8859-1')
file_output = codecs.open("iso-8859-1.txt"+"b", 'w', 'utf-8')
l=file_stream.readline()
    file_output.write(l)
file_stream.close()
file_output.close()
print time()-t1
```

File output: (with "b" appended to it's name)

Olé

Figure 3.1.4. – Output of Python 2.0 Transcoding example with writing output to a file.

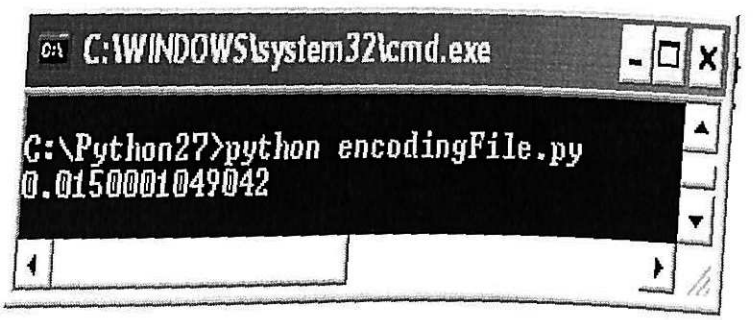


Table 3.1.5. - Ruby Transcoding example with writing output to a file.

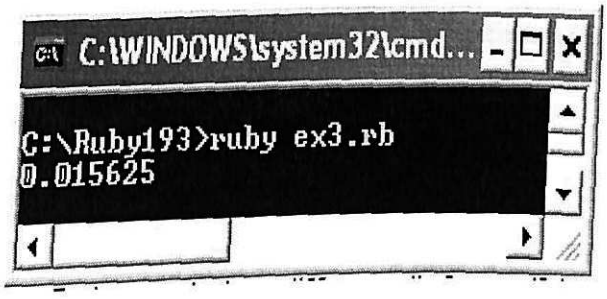
Code line
t1=Time.now()

```
f = File.open("iso-8859-1.txt", "r:iso-8859-1")
f1 = File.open("utf-8.txt", "w:utf-8")
line = f.gets
f1.puts line
puts Time.now()-t1
```

File output:

Olé

Figure 3.1.5. – Output of Ruby Transcoding example with writing output to a file.



Comparison:

Code lines: Python: 6 lines, Ruby: 5 lines

Time for executing: Python time is less than Ruby time

Note: Python required two different calls for specifying input and output encoding of file. But Ruby take one line for specifying reading encoding and writing encoding system. Also in Ruby there're additional two functionalities for outputting names of encoding.

Python 3.0 uses the concepts of *text* and (binary) *data* instead of Unicode strings and 8-bit strings. All text is Unicode; however *encoded* Unicode is represented as binary data. The type used to hold text is str, the type used to hold data is bytes. The biggest difference with the 2.x situation is that any attempt to mix text and data in Python 3.0 raises TypeError, whereas if you were to mix Unicode and 8-bit strings in Python 2.x, it would work if the 8-bit string happened to contain only 7-bit (ASCII) bytes, but you would get UnicodeDecodeError if it contained non-ASCII values. This value-specific behavior has caused numerous sad faces over the years. [14]

Because of the specificity of language in Python 3.0 the output is given in hex code where the utf-8 coding occures and at the start there's a letter 'b' which indicates that this is a binary output. In Python 2.0. the output is more closer to string representation but also not é. And the true representation of e apostrophe comes when writing to a file with "b" appended to the file name.

In Ruby it take less (4) lines and the output is in normal txt format displays unicode string "olé"

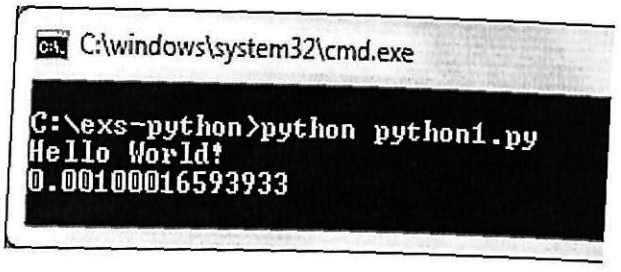
3.2. Executing statements

Python statement example:

Table 3.2.1. - Statement in Python

Code line
from time import time t1=time() print("Hello World!") print time()-t1

Figure 3.2.1. – Output of Statement in Python example

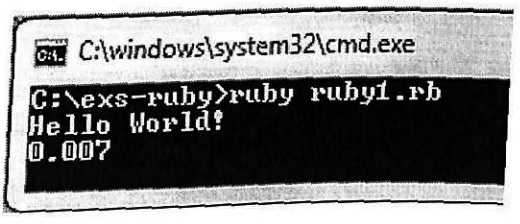


The same example in Ruby:

Table 3.2.2. - Statement in Ruby

Code line
t1=Time.now
puts "Hello World!"
puts Time.now-t1

Figure 3.2.2. Output of Statement in Ruby



Comparison:

Code lines: Python: 4 lines, Ruby: 3 lines

Time for executing: Python time is less than Ruby time

Note: Actually in Python it was import statement taking one more line. And because of this kind of statement not required in Ruby rubiests claim it to be totally object oriented.

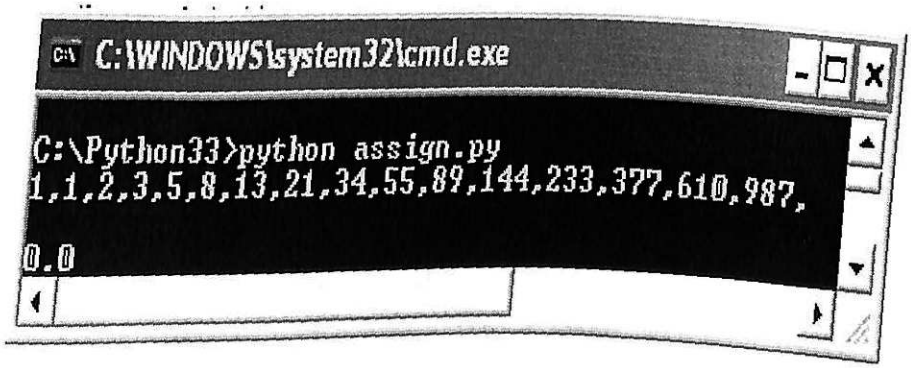
3.3. Operator Expressions

Operator Expression example in Python:

Table 3.3.1. - Operator expression in Python

Code line
from time import time t1=time() a, b = 0, 1 while b < 1000: print(b, end=',') a, b = b, a+b print time()-t1

Figure 3.3.1. – Output of operator expression in Python example



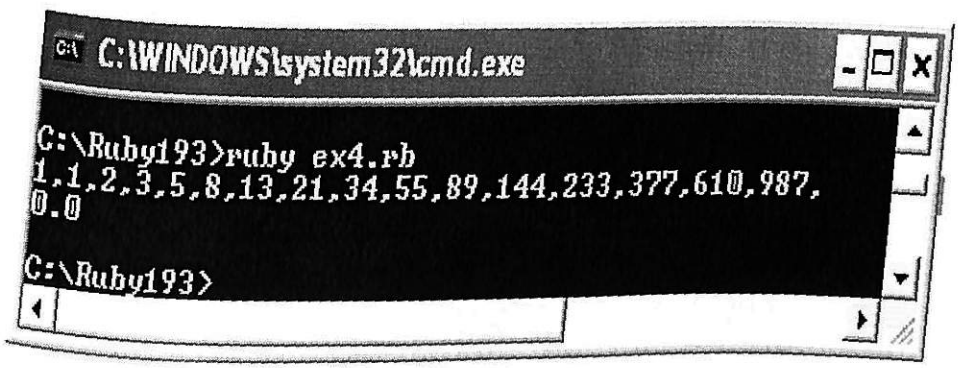
Ruby defining:

Table 3.3.2. - Operator expression in Ruby

Code line

t1=Time.now
a, b = 0, 1
while b < 1000
  print b,","
  a, b = b, a+b
end
puts "\n",Time.now-t1

Figure 3.3.2. – Output of operator expression in Ruby



Comparison:

Code lines for defining method: Python: 4 lines, Ruby: 5 lines

Time: Python time is same as Ruby time

Note: In both languages parallel assignment is available.

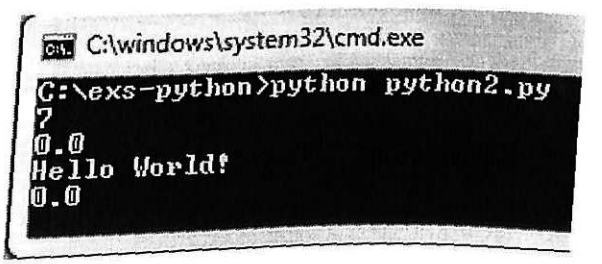
3.4. Method / Function defining

Method and function example in Python:

Table 3.4.1. - Method and function in Python

Code line
def sum(n1,n2): return n1+n2 from time import time t1=time() print(sum(3,4)) print time()-t1 t1=time() print(sum("Hello ","World!")) print time()-t1

Figure 3.4.1. – Output of Method and function in Python

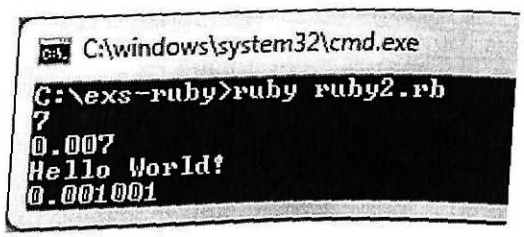


Ruby defining:

Table 3.4.2. - Method and function in Ruby

Code line
def sum(n1,n2) n1+n2 end t1=Time.now puts sum(3,4) puts Time.now-t1 t1=Time.now puts sum("Hello ","World!") puts Time.now-t1

Figure 3.4.2. – Output of Method and function in Ruby



Comparison:

Code lines for defining method: Python: 2 lines, Ruby: 3 lines

Time: Python time is less than Ruby time

Note: Ruby needs additional end statement to finish defining and Python needs return keyword to be processed like a function not procedure. Without return or return without arguments will give us None result in Python. And in Ruby every called method returns a value (although no rule says you have to use that value).The value of a method is the value of the last statement executed during the method's execution. Also in Ruby because it looks at everything as objects nil argument calling will give undefined method error for NilClass as in Python calling with None argument.

3.5. IF Statement and Standard input, gets

Python IF statement:

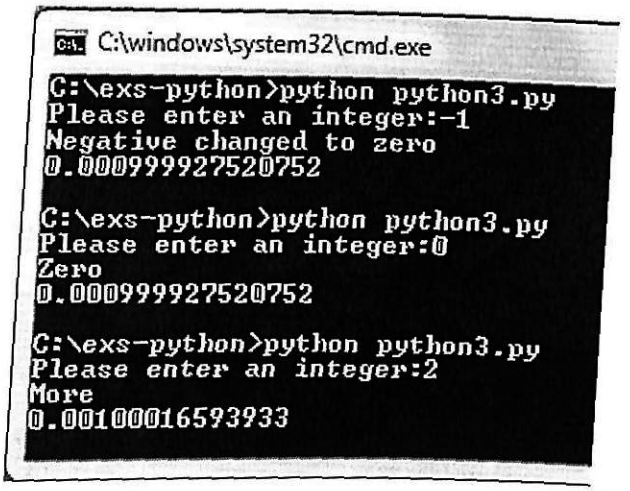
Table 3.5.1. - if statement in Python

Code line

from time import time
x=int(input("Please enter an integer:"))
t1=time()
if x<0:
  x=0
  print('Negative changed to zero')
elif x==0:
  print('Zero')
elif x==1:
  print('One')

```
else:
    print('More')
print time()-t1
```

Figure 3.5.1. – Output of if statement in Python



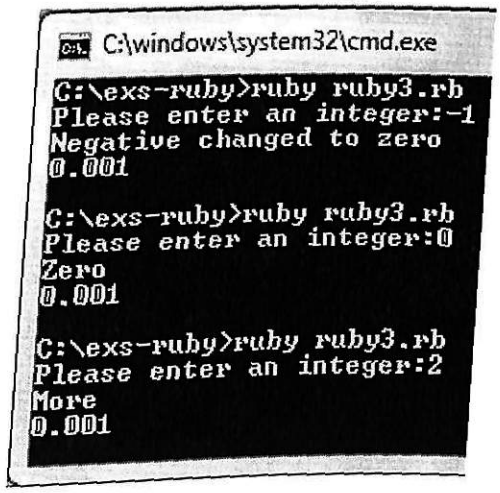
Ruby IF statement:

Table 3.5.2. If statement in Ruby

Code line
print "Please enter an integer:" x=Integer(gets) t1=Time.now if x<0 x=0

```
        print("Negative changed to zero\n")
    elsif x==0
        print "Zero\n"
    elsif x==1
        puts 'One'
    else
        puts 'More'
    end
    puts Time.now-t1
```

Figure 3.5.2. – Output of If statement in Ruby



Comparison

Code lines for input, gets: Python: 1 line, Ruby: 2 lines

Code lines for IF statement: Python: 9 lines, Ruby: 10 lines

Time: Almost same

Note: In Python after the beginning of any statement we need to put colon. In Ruby there is no colon to be put and the brackets of any methods are optional if after method call there are no operations on return values of methods. In Ruby the brackets of print statement is must with one argument, with more arguments it must be omitted.

3.6. The logic of Range

Python's range() function:

Table 3.6.1. – Python 3.3's range() example.

Code line
from time import time t1=time() print(list(range(10))) print(list(range(5,10))) print(list(range(0,10,3))) print(list(range(-10,-100,-30))) print(time()-t1)

Figure 3.6.1. – Output of Python 3.3's range() example.

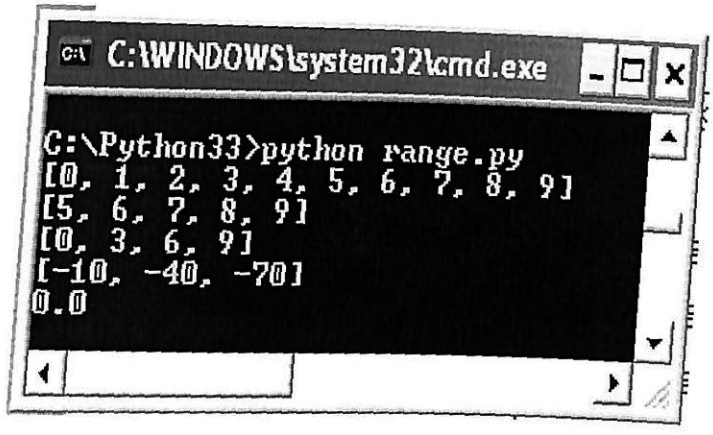


Table 3.6.2. – Python 2.7's range() example.

Code line
from time import time
t1=time()
print(range(10))
print(range(5,10))
print(range(0,10,3))
print(range(-10,-100,-30))
print(time()-t1)

Figure 3.6.2. – Output of Python 2.7's range() example.

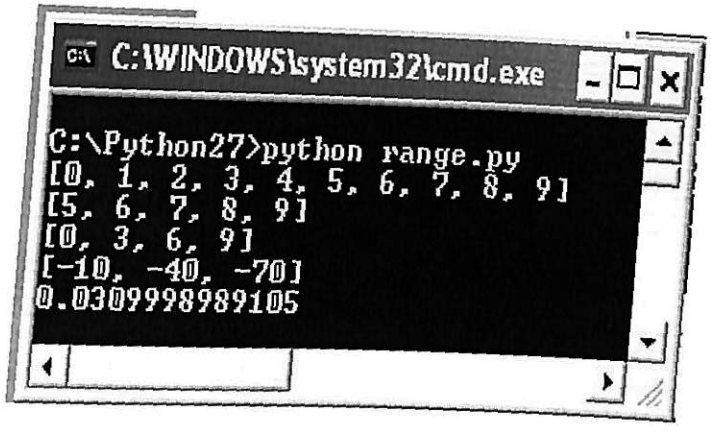
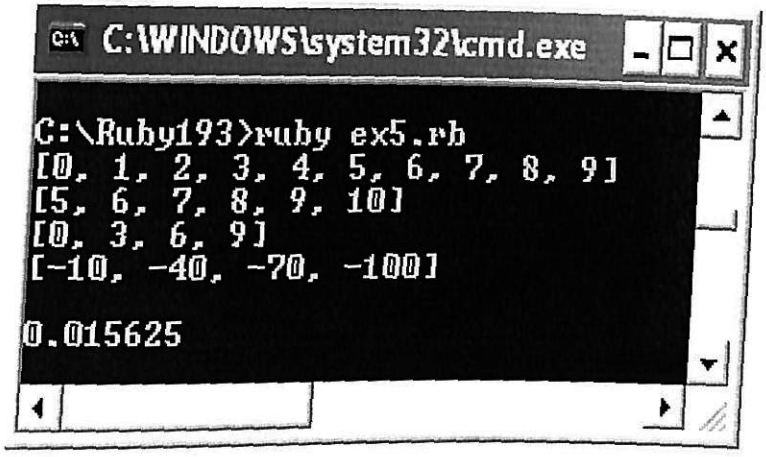


Table 3.6.3. - Ruby's Range object.

Code line
t1=Time.now
print (0...10).to_a,"\n"
print (5..10).to_a,"\n"
print (0..10).step(3).to_a,"\n"
print (-100..-10).step(30).to_a.reverse,"\n"
puts "\n",Time.now-t1

Figure 3.6.3 – Output of Ruby's Range object.



Comparison:

Code lines: Python: 4 lines, Ruby: 4 lines

Time: Python2.7 is slower than Python 3.3, but Python 3.3 is faster than Ruby

Note: In Python's 2.7 version the built-in function range() would give list of iterables directly to output, but in version 3.3 we have to call function list() which converts iterables into list. In Ruby also iterables should be converted to array.

The logic of version 3.3 is the same as version 2.7's xrange().

The syntax difference of Ruby comes because it's Range is an object. And the syntax ".." means to include the last element into range while "..." means to exclude the last element.

In Python the step argument may take positive and negative arguments, but Ruby allows only positive argument to the step() method of Range object.

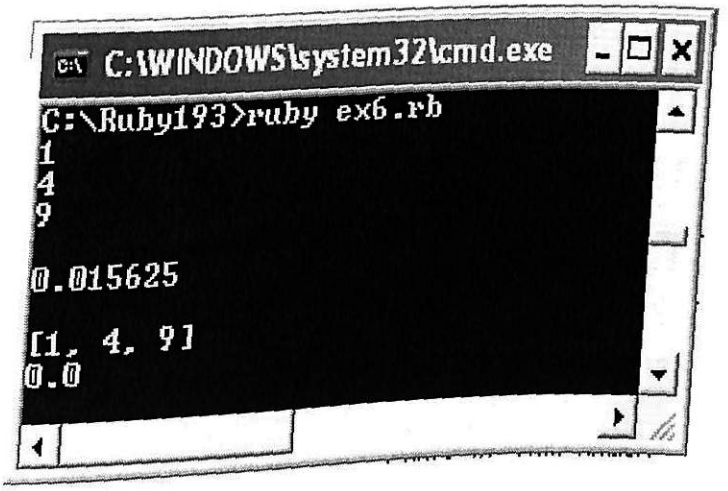
3.7. Iterators of Ruby Vs List Comprehension of Python

Ruby example:

Table 3.7.1. – Iterators of Ruby example.

Code line
t1=Time.now [1,2,3].each { |x| print x*x,"\n" } puts "\n",Time.now-t1 t1=Time.now print "\n",[1,2,3].collect { |x| x*x } puts "\n",Time.now-t1

Figure 3.7.1. – Output of Iterators of Ruby example.

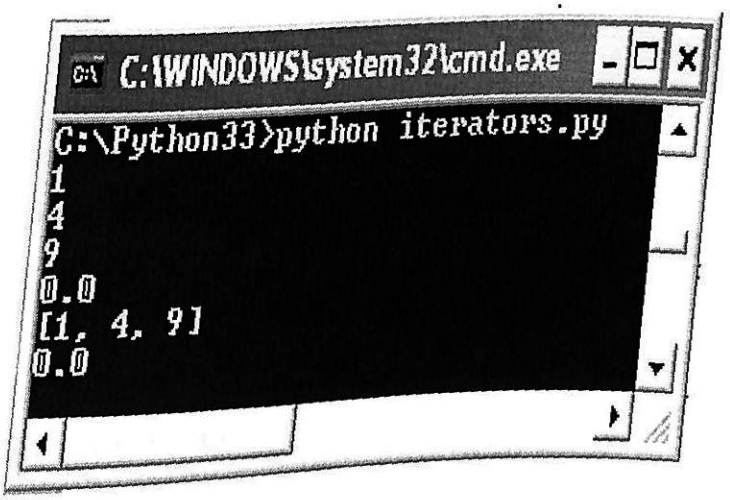


Python example:

Table 3.7.2. - List Comprehension of Python example.

Code line
from time import time t1=time() for x in [1,2,3]:   print(x*x) print(time()-t1) t1=time() print([x*x for x in [1,2,3]]) print(time()-t1)

Figure 3.7.2. – Output of List Comprehension of Python example.



Comparison:

Code lines: Python: 2 and 1 lines, Ruby: 1 line in both

Time: Almost same.

Note: Due to the power of blocks, Ruby supports constructs called iterators. An iterator goes over every element of any container class and may do any processing on it.

Python achieve a similar effect by using list comprehensions.[1]

3.8. Data Structures

As data structures in Ruby Arrays and Hashes are used, in Python Tuples and dicts are used. The operation time consuming is almost same in both of them, the only difference occurs when it comes to the additional functionality availability via methods in their libraries. To see them you can go to the section Additional-1.

Ruby has more methods than Python to work with these data structures.

3.9. All constructs have a value

In Ruby the value can be returned from any construct, but in python it's not at that extent. Python lacks the complex conditional construct evaluation to assign it directly to a variable, but the simple conditions can be made and assigned. Let's look it in an example.

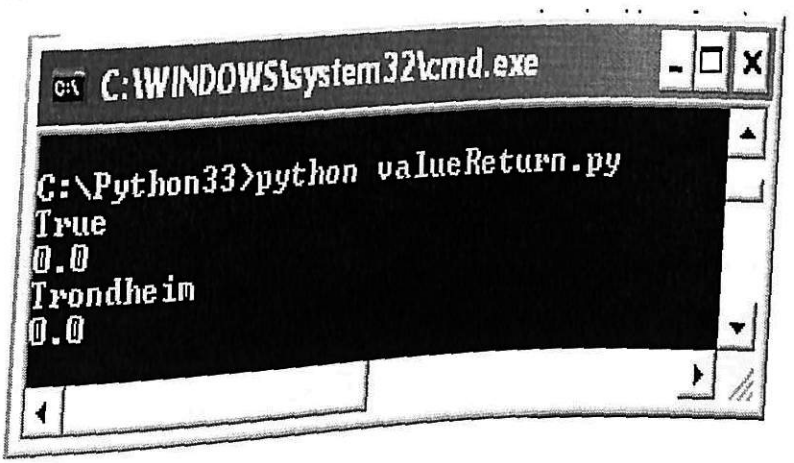
Python example:

Table 3.9.1. – Python's conditional construct

Code line
from time import time t1=time() txt = "Hello World"

```
a = len(txt) > 3
print (a)
print(time()-t1)
t1=time()
string1, string2, string3 = "", "Trondheim", "Hammer Dance"
non_null = string1 or string2 or string3
print (non_null)
print(time()-t1)
```

Figure 3.9.1. – Output of Python's conditional construct



And the Ruby example is as in Part Two of this work.

3.10. Exception handling

In both languages raising user defined errors is done with the keyword *raise*. And the syntax for handling them is as follows in the examples below:

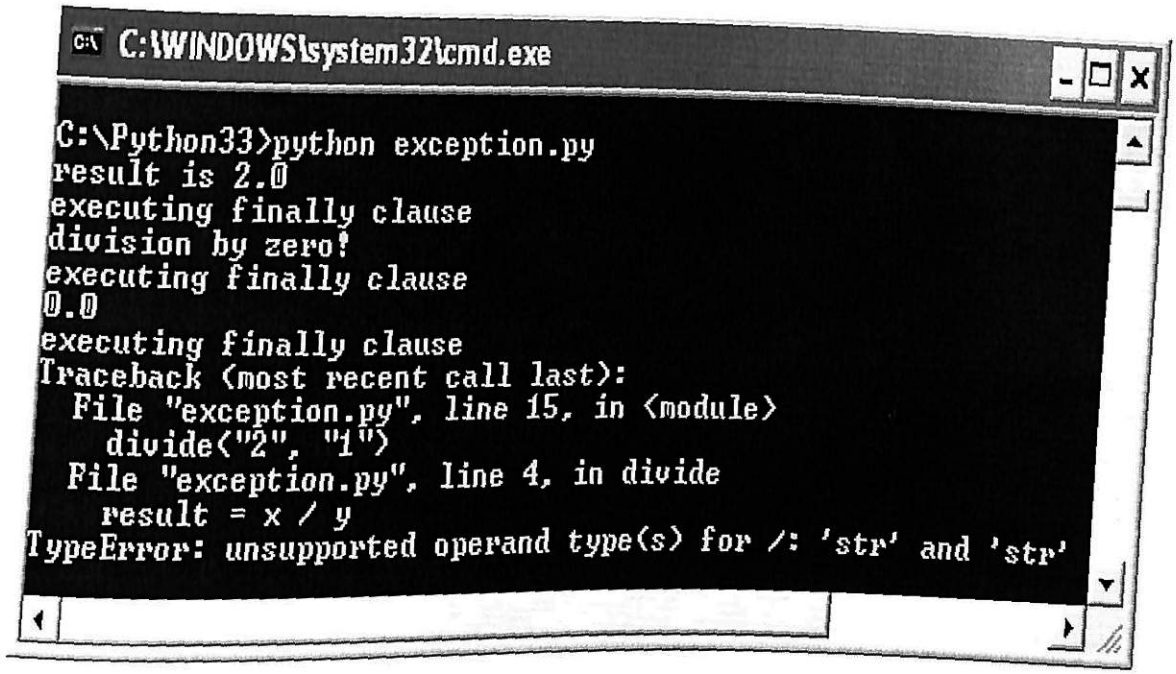
Python example:

Table 3.10.1. – Python's exception handling

Code line

from time import time
def divide(x, y):
  try:
  result = x / y
  except ZeroDivisionError:
  print("division by zero!")
  else:
  print("result is", result)
  finally:
  print("executing finally clause")
t1=time()
divide(2, 1)
divide(2, 0)
print(time()-t1)
divide("2", "1")

Figure 3.10.1. – Output of Python's exception handling



Ruby example:

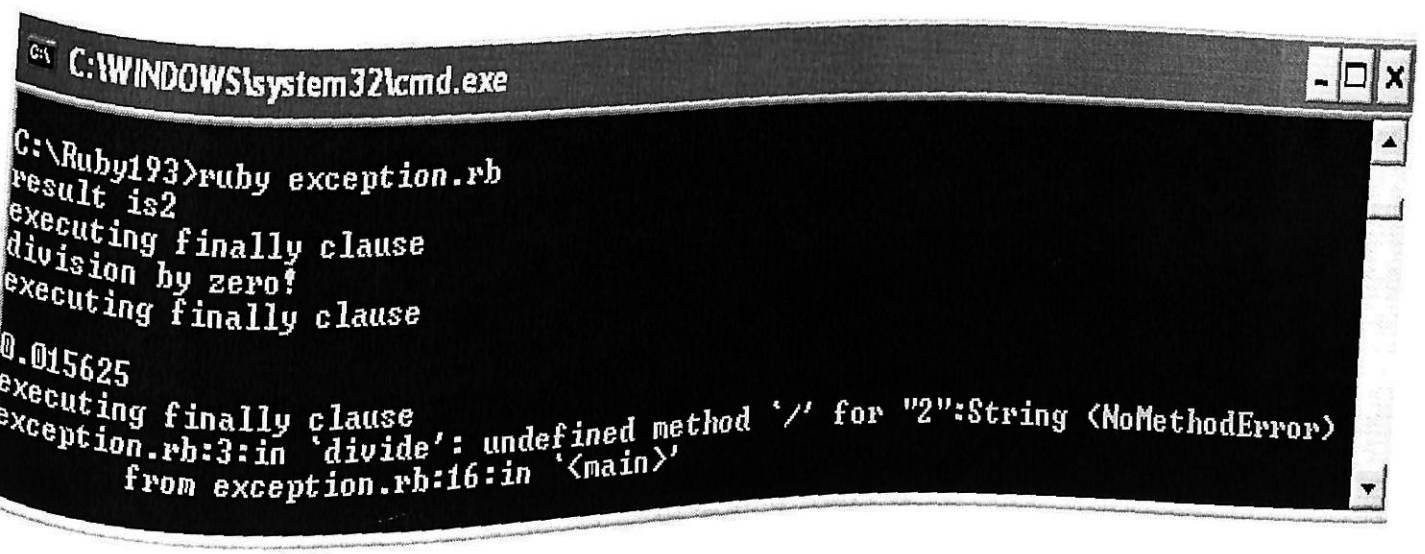
Table 3.10.2. – Ruby's exception handling

Code line

def divide(x, y)
begin
result = x / y
rescue ZeroDivisionError
print("division by zero!\n")
else
print("result is", result,"\n")

```
    ensure
        print("executing finally clause\n")
    end
end
t1=Time.now
divide(2, 1)
divide(2, 0)
puts "\n",Time.now-t1
divide("2", "1")
```

Figure 3.10.2. – Output of Ruby's exception handling



Comparison:

Code lines: Python: 8 lines, Ruby: 9 lines

Time: Python time is less than Ruby.

Note: Everything in both is done same way, but the keywords for handling exceptions are different. Ruby's *begin* and Python's *try* start the handling block. Ruby's *rescue* and Python's *except* catch the Exception needed. In both *else* clause is used for normal execution in case no error occurs. And for the last step to execute in any case of outcome the clause is defined by Python's *finally* and Ruby's *ensure* keywords. So the difference is just in the naming of the keywords, but the mechanism is done in the same way.

3.11. Regular expressions

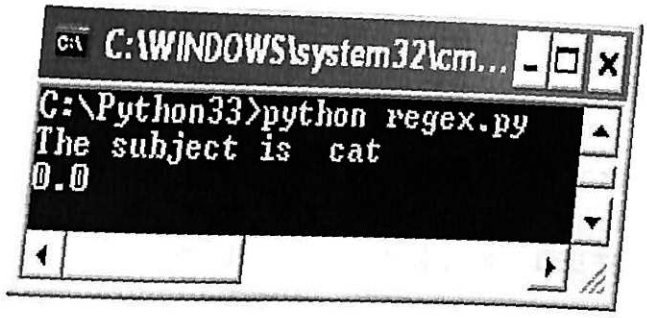
Python's regex module was the first to offer a solution: named capture. By assigning a name to a capturing group, you can easily reference it by name. (?P<name>group) captures the match of group into the backreference "name". You can reference the contents of the group with the numbered backreference \1 or the named backreference (?P=name). [12] And the same logic works in Ruby too.

Python example:

Table 3.11.1. – Python's regular expressions

Code line
from time import time
import re
t1=time()
sentence = re.compile(r'(?P<subject>cat|dog|gerbil)')
md = sentence.search("The cat drinks water")
if md:
print("The subject is ",md.groupdict()['subject'])
print(time()-t1)

Figure 3.11.1. – Output of Python's regular expressions

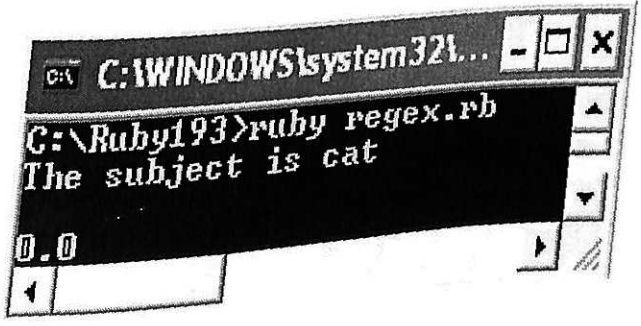


Ruby example:

Table 3.11.2. – Ruby's regular expressions

Code line
t1=Time.now
sentence = %r{(?<subject>cat| dog| gerbil)}
md = sentence.match("The cat drinks water")
puts "The subject is #{md[:subject]}"
puts "\n",Time.now-t1

Figure 3.11.2. – Output of Ruby's regular expressions



Comparison:

Code lines: Python: 5 lines, Ruby: 3 lines

Time: Almost same.

Note: Python takes more lines because of syntax construction and the import statement. And it must be noted that from Python to Ruby it's easy to convert the code, but hard to do otherwise when it comes to complicated regular expressions. It's because Ruby syntax differs from Python's.

3.12. Garbage collections

Garbage collection is the automated process of reclaiming memory allocated by unused objects. When an object is not referred to by any other objects, it's considered "garbage," not part of the state of the program any longer. Its memory should be returned to the pool of free memory to be used by other objects or returned to the operating system.

There was a time when garbage collection was considered too heavy. Some languages used it, but the overhead of a garbage collector was certainly a burden. In these languages, such as C or C++, memory must be manually allocated from the heap and, when you're finished with it, manually returned to the operating system. This is extremely efficient as there is little to no overhead, but it's not without its problems.

If a program allocated memory but forgot to deallocate it, it remain allocated until the program ends (and all its memory is returned to the operating system). This is called a *memory leak*and often cause programs to continually allocate more memory, consuming more and more memory until the operating system kills the process. This class of bug is extremely difficult to track down, and is one of the biggest reasons of moving to higher level languages.[15]

The working mechanisms are same in both languages, but the implementation differs. In Ruby GC is a class object with it's methods, but in Python "gc" is an interface with functions in it. So just to make a review of them look in the section Additional-2.

Here it must be noted that Ruby GC's methods are a bit less than the functic given in Python module.

Python's memory allocation and deallocation method is automatic. The us does not have to preallocate or deallocate memory by hand as one has to when usin dynamic memory allocation in languages such as C or C++. Python uses tw strategies for memory allocation **reference counting** and **garbage collection**.[16]

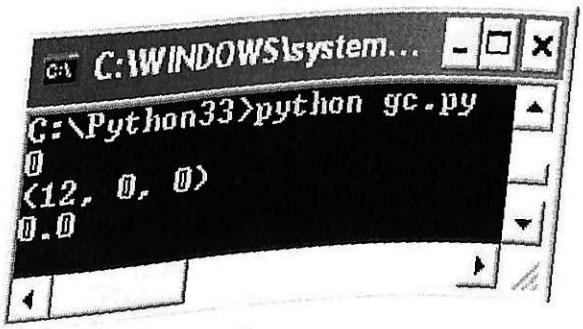
Let's take a look in an example:

Python example:

Table 3.12.1. – Python's garbage collection

Code line
from time import time
import gc
t1=time()
gc.enable()
print(gc.collect())
print(gc.get_count())
print(time()-t1)

Figure 3.12.1. – Output of Python's garbage collection

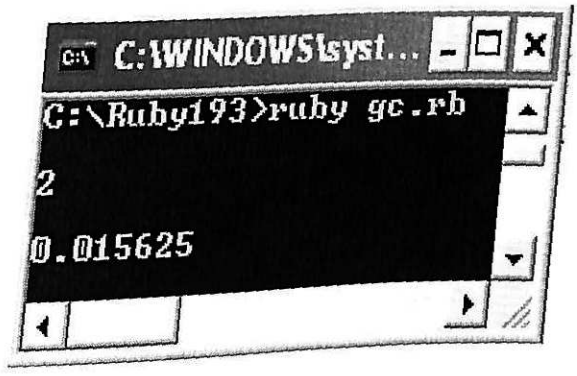


Ruby example:

Table 3.12.2. – Ruby's garbage collection

Code line
t1=Time.now
GC.enable
puts GC.start
puts GC.count
puts "\n",Time.now-t1

Figure 3.12.2. – Output of Ruby's garbage collection



Comparison:

Code lines: Python: 4 lines, Ruby: 3 lines

Time: Python time is less than the Ruby.

Note: Python takes more lines because of the import statement

3.13. Multi-threading

Running several threads is similar to running several different programs concurrently, but with the following benefits:

•Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.

•Threads sometimes called light-weight processes and they do not require much memory overhead; theycare cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

•It can be pre-empted (interrupted)

•It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

For both languages let's make an example:

Python example:

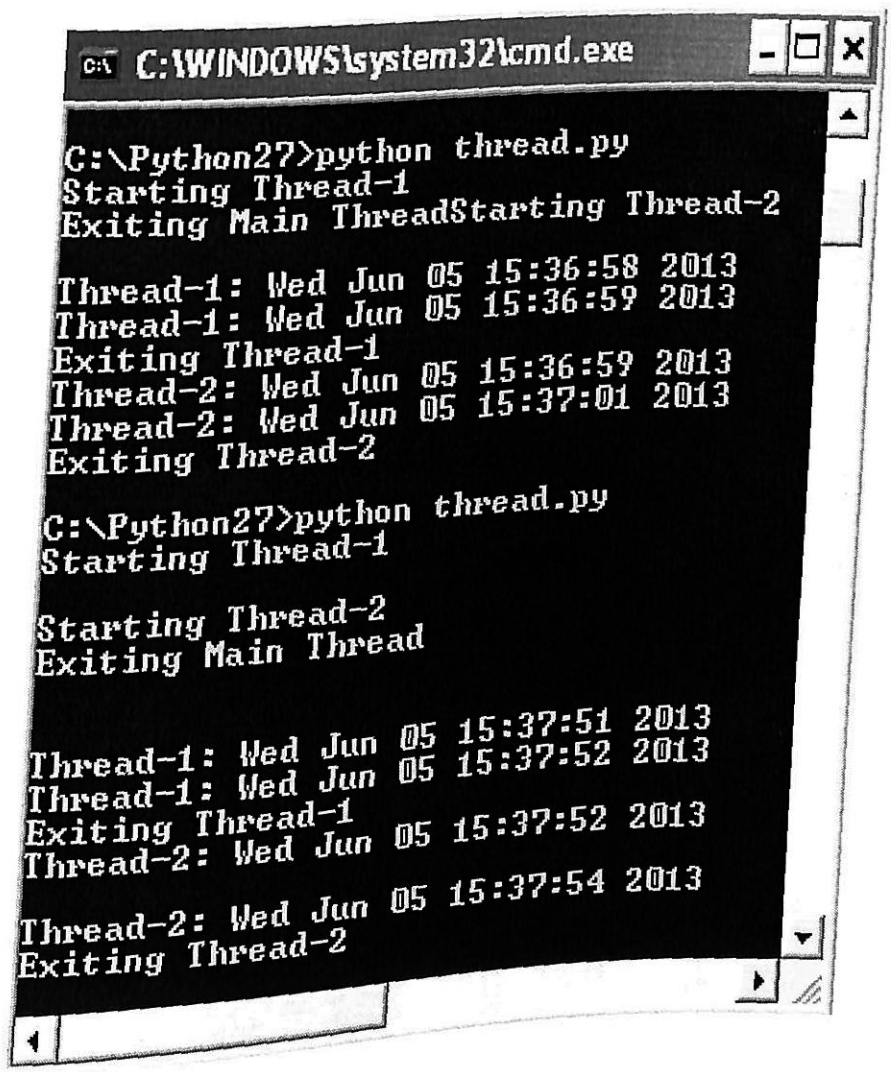
Table 3.13.1. – Python's multi-threading

Code line
import threading
import time
exitFlag = 0
class myThread (threading.Thread):
def __init__(self, threadID, name, counter):

```
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name+"\n"
        print_time(self.name, self.counter, 2)
        print "Exiting " + self.name+"\n"


def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1


# Create new threads
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)


# Start new Threads
```

```
thread1.start()

thread2.start()


print "Exiting Main Thread\n"
```

Figure 3.13.1. – Output of Python's Multithreading



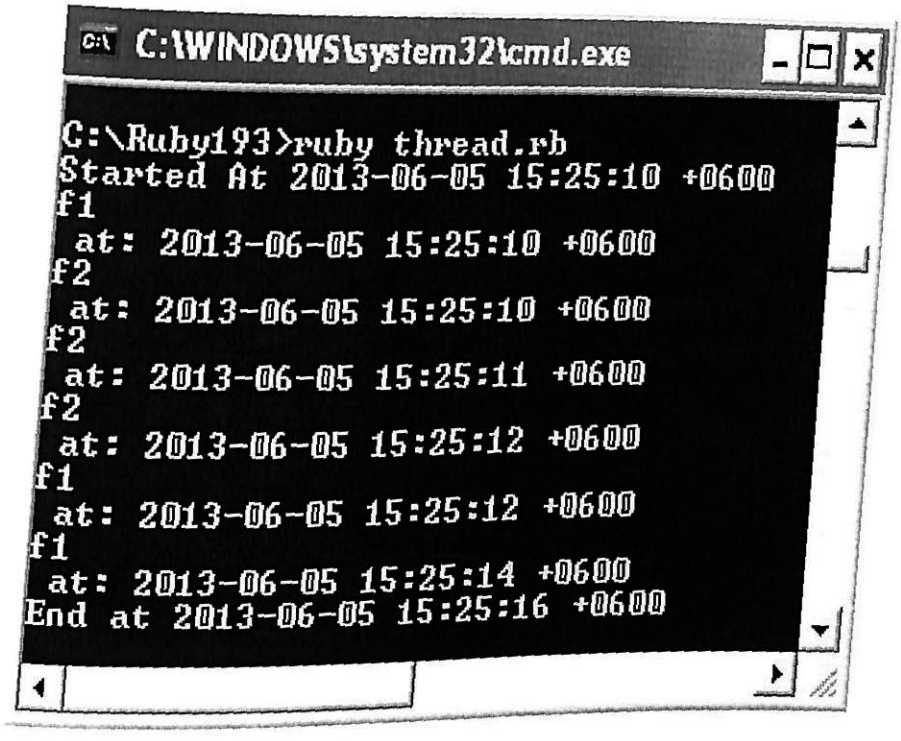
Ruby example:

Table 3.13.2. – Ruby's multithreading

Code line

def func(delay,name)
  i=0
  while i<=2
  puts name," at: #{Time.now}"
  sleep(delay)
  i=i+1
  end
end
puts "Started At #{Time.now}"
t1=Thread.new{func(2,"f1")}
t2=Thread.new{func(1,"f2")}
t1.join
t2.join
puts "End at #{Time.now}"

Figure 3.13.2. – Output of Ruby's multithreading



Comparison:

Code lines: Python: 19 lines, Ruby: 9 lines

Time: It take same duration for both.

Note: Here only the main declaring part for threading was count. As we can see Ruby's object approach make it easier to declare threading.

3.14. *Object Oriented Programming*

Let's discuss this approach by giving examples.

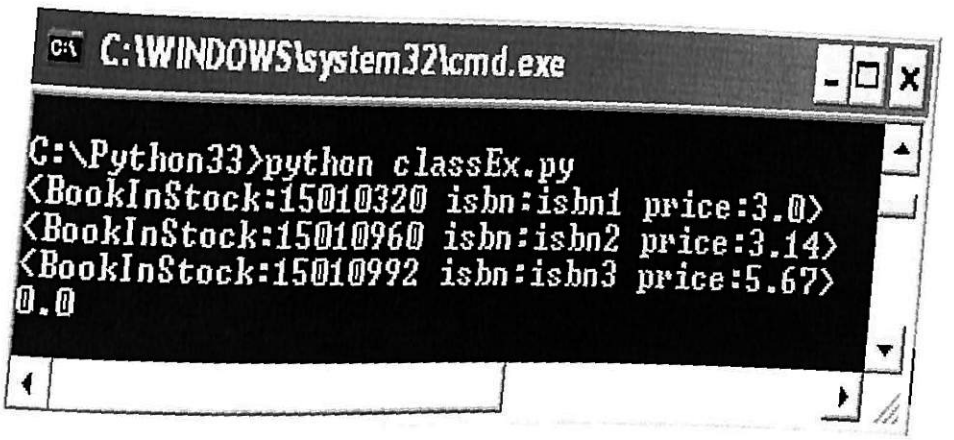
Python example:

Table 3.14.1. – Python's Class example.

Code line

```
from time import time
t1=time()
class BookInStock:
    def __init__(self,isbn,price):
        self._isbn = isbn
        self._price = float(price)
    def __repr__(self):
        return "<BookInStock:%s isbn:%s price:%s>" %(id(self),self._isbn,
self._price)
b1 = BookInStock("isbn1", 3)
print(b1)
b2 = BookInStock("isbn2", 3.14)
print(b2)
b3 = BookInStock("isbn3", "5.67")
print(b3)
print(time()-t1)
```

Figure 3.14.1. Output of Python's Class example



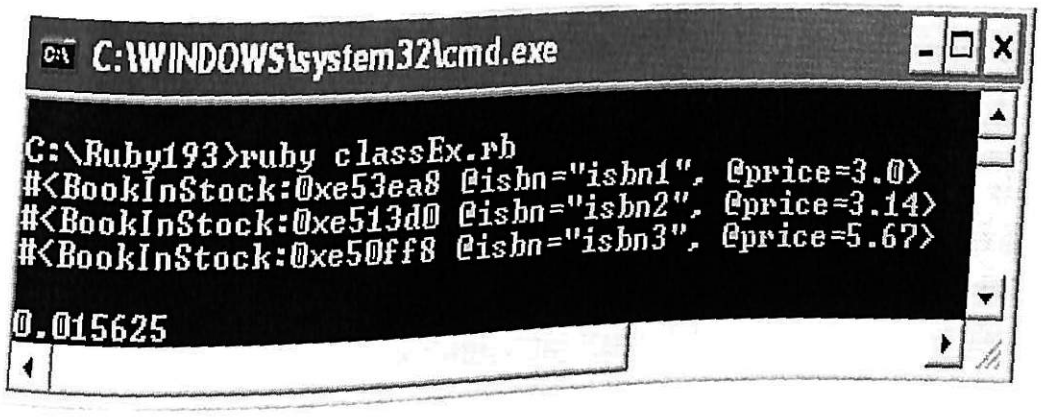
Ruby example:

Table 3.14.2. Ruby's Class example

Code line
t1=Time.now
class BookInStock
def initialize(isbn, price)
@isbn = isbn
@price = Float(price)
end
end
b1 = BookInStock.new("isbn1", 3)
p b1
b2 = BookInStock.new("isbn2", 3.14)

```
p b2
b3 = BookInStock.new("isbn3", "5.67")
p b3
puts "\n",Time.now-t1
```

Figure 3.14.2. – Output of Ruby's Class example



Comparison:

Code lines: Python: 6 lines, Ruby: 6 lines

Time: Python time is less than Ruby time.

Note: Here only the main declaring part for class was count as code lines.

In Ruby the last extra two lines were because of the keyword *end* to close the declaring of class and the method. In Python last two lines were because of the method *__repr__* to make an appropriate representation for printing out. And also the predefined methods for classes in Python are written with two underlines before and after the keyword itself.

In Ruby *self* doesn't need to be shown explicitly, but Python requires it always be written there.

4. Conclusion

In this work basic comparison between two scripting languages Ruby and Python was made. This preview will make it possible for your further progress in introduction with these two languages.

Larry Wall, the designer of Perl, has the slogan "There's more than one way to do it". In contrast, Bertrand Meyer, the designer of Eiffel, says "A programming language should provide one good way of performing any operation of interest; it should avoid providing two." Ruby follows the anarchist approach of Larry Wall, while Python follows the bondage-and- discipline approach of Eiffel.[13]

Both languages are High Level, Garbage Collected, and Dynamically Typed. Both provide an interactive shell, standard libraries, and persistence support.

So far Pythonists have emphasized their language's *ease through consistency*, extensive libraries, docs, etc, and argued that Ruby's advantages in elegance are overstated or nonexistent. Rubyists have stressed their language's *ease through conceptual elegance*, slightly greater depth of OO, the power of Blocks In Ruby, and the nice feeling you get from doing things the Ruby Way.

A few others have objected that the whole idea of being a "fan" or "-ist" of a language is very silly.

Even though there are considerations that must be taken into account.

The main Ruby point that while claiming it to be totally object oriented it allows developers make changes in the existing classes. This means if there is a class A already existing in Ruby specification, developer-1 makes a new class B deriving from it and overrides some existing methods by changing the logic in which it works. So if a developer-2 comes and tries to make things as specified in specification the error will be thrown. Even if these changes will occur in a newly declared class C the same outcome results. Assume Class C newly declared by any developer X and some other developer Y makes changes in it, the developer X won't be able to use class C in the way it was previously declared.

As we know already Python doesn't allow any changes in already defined classes.

Taking these two facts into account we can conclude that Python is a better choice for using in big projects were things must work as they're assumed to be, so that no one will be confused while continuing the frozen work. But Ruby will be good enough in usage of small projects, otherwise if it's a big project then there must be already defined set of rules, conventions for all developers to be used while programming.

Ruby is good for hacking as it makes coding fun for a developer coming with any background experience. Python is good enough for conservative programming. That is to say if you want to use Ruby in right way you must have very good experience of programming with different languages, but Python doesn't require to know any language to start programming without fatal errors.

As we know it's rare to meet the good experienced developer and the projects are set in a group of developers. The chance that in one team all will be well experienced is small, because while forming a team for project there are many aspects like time and money to be evaluated. If it's an urgent project (in nowadays there can be said nothing exists which isn't urgent for our now existing world), we can't just wait and search for a qualified developer. Because of this fact Python is more popular in big project development than Ruby.

In this work also it must be noted that compiler of Ruby was weird while compared to Python's. Because in testing written codes sometimes compiler would just give the line number where syntax error occurs and not specify clearly what the error would be. This must be because of the principle "there is more than one way to do it". Python's conventions on coding are strict so the developer won't be confused if the compiler doesn't specify the error clearly. He just has to follow the specifications of python coding.

In conclusion we can just say that it's a choice of a developer according to the project size, performance needs, distribution availability specifications and etc. This work was a slightly overview of language differences between Ruby and Python.

References

1. "A comparison of object oriented scripting languages: Python and Ruby". Kaustub D. kd@cs.washington.edu, David Grimes grimes@cs.washington.edu, December 18, 2001. Page 2.
2. "Scripting: higher level programming for the 21st Century" Ousterhout, John K. , Volume: 31 , Issue: 3, Page(s): 23 - 30 , Product Type: Journals & Magazines: Computer.
3. "The Making of Python". Artima Developer. Retrieved 2007-03-22.
4. "TIOBE Programming Community Index for March 2012". TIOBE Software. March 2012. http://www.tiobe.com/index.php/content/paperinfo/tpci/. Retrieved 25 March 2012.
5. http://wiki.python.org/moin/AppsWithPythonScripting*
6. http://www.ipa.go.jp/about/press/20120402_2.html*
7. The Philosophy of Ruby, A Conversation with Yukihiro Matsumoto, Part I by Bill Venners on 2003-09-29 (Artima Developer): http://www.artima.com/intv/ruby4.html*
8. Programming Ruby, The Pragmatic Programmers' Guide, Dave Thomas with Chad Fowler and Andy Hunt, Second Edition.
9. Python Tutorial, Release 3.2.3., Guido van Rossum, Fred L.Drake, Jr., September 2012.
10. http://en.wikipedia.org/wiki/History_of_Python*
11. http://www.sitepoint.com/typing-versus-dynamic-typing/_*
12. http://www.regular-expressions.info/named.html_*
13. http://www.ianbicking.org/re-ruby-and-python-compared.html_*
14. http://docs.python.org/3/whatsnew/3.0.html*

15. http://ruby.about.com/od/advancedruby/ss/Garbage-Collection.htm_*

16. http://www.digi.com/wiki/developer/index.php/Python_Garbage_Collection_*

17. http://www.tutorialspoint.com/python/python_multithreading.htm_*

18. http://rosettacode.org/wiki/ISO/IEC_30170_*

19. http://www.tiobe.com/index.php/content/paperinfo/tpci/tpci_definition.htm_*

*Accessed on 06/06/2013

Additional-1

A. Python's Sequence Types — list, tuple, range

There are three basic sequence types: lists, tuples, and range objects.

A.1. Common Sequence Operations

The operations in the following example are supported by most sequence types both mutable and immutable.

Figure A.1.1. – Usage examples of sequence operations in Python.

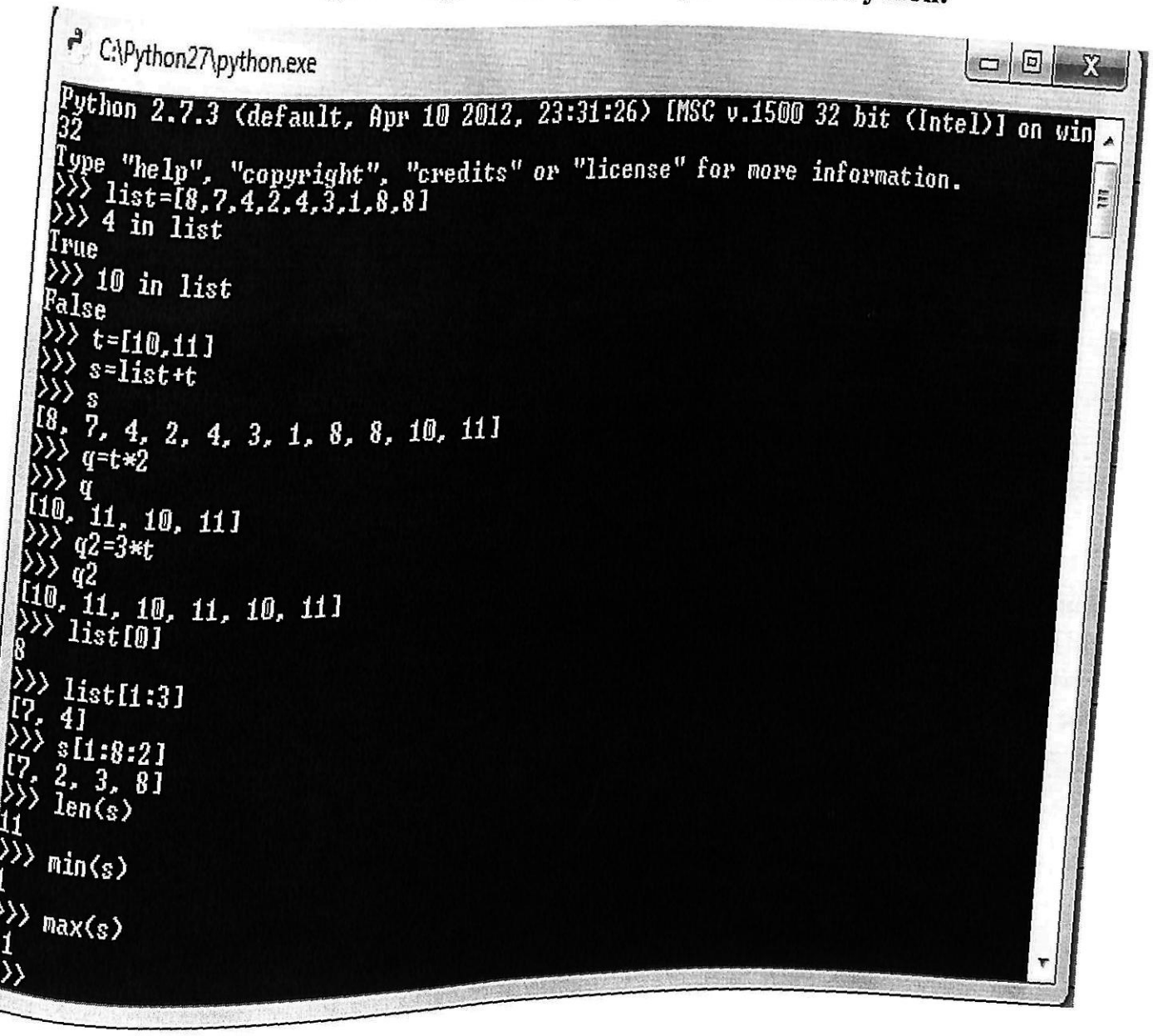
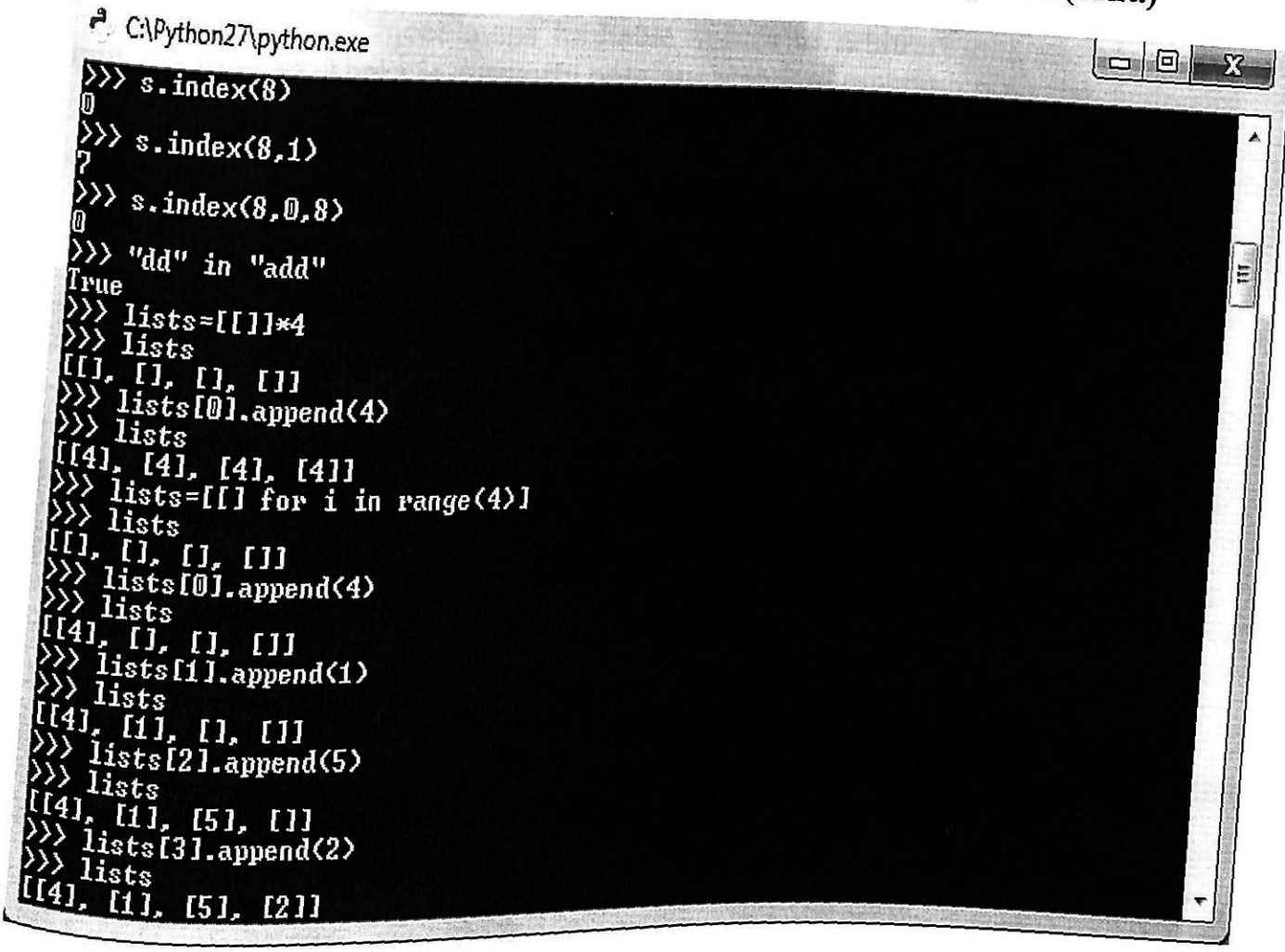


Figure A.1.2. - Usage examples of sequence operations in Python.(cont.)



Sequences of the same type also support comparisons. In particular, tuples and lists are compared lexicographically by comparing corresponding elements. This means that to compare equal, every element must compare equal and the two sequences must be of the same type and have the same length. (For full details see comparisons in the language reference.)

B. Python's Mapping Types — dict

A mapping object maps hashable values to arbitrary objects. Mappings are mutable objects. There is currently only one standard mapping type, the dictionary.

To illustrate, the following examples all return a dictionary equal to {"one": 1, "two": 2, "three": 3}:

```
>>> a = dict(one=1, two=2, three=3)

>>> b = {'one': 1, 'two': 2, 'three': 3}

>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))

>>> d = dict([('two', 2), ('one', 1), ('three', 3)])

>>> e = dict({'three': 3, 'one': 1, 'two': 2})

>>> a == b == c == d == e

True
```

Providing keyword arguments as in the first example only works for keys that are valid Python identifiers. Otherwise, any valid keys can be used.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

classmethod fromkeys(seq[, value]) -Create a new dictionary with keys from seq and values set to value. *fromkeys()* is a class method that returns a new dictionary. value defaults to None.

get(key[, default]) -Return the value for key if key is in the dictionary, else default. If default is not given, it defaults to None, so that this method never raises a KeyError.

items() -Return a new view of the dictionary's items ((key, value) pairs). See the documentation of view objects.

keys() -Return a new view of the dictionary's keys. See the documentation of view objects.

pop(key[, default]) -If key is in the dictionary, remove it and return its value, else return default. If default is not given and key is not in the dictionary, a KeyError is raised.

popitem() -Remove and return an arbitrary (key, value) pair from the dictionary. *popitem()* is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling popitem() raises a KeyError.

setdefault(key[, default]) -If key is in the dictionary, return its value. If not, insert key with a value of default and return default. *default* defaults to None.

update([other]) -Update the dictionary with the key/value pairs from other, overwriting existing keys. Return None. *update()* accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: d.update(red=1, blue=2).

values() -Return a new view of the dictionary's values.

Figure B.1. – Operations that dictionaries support in Python.

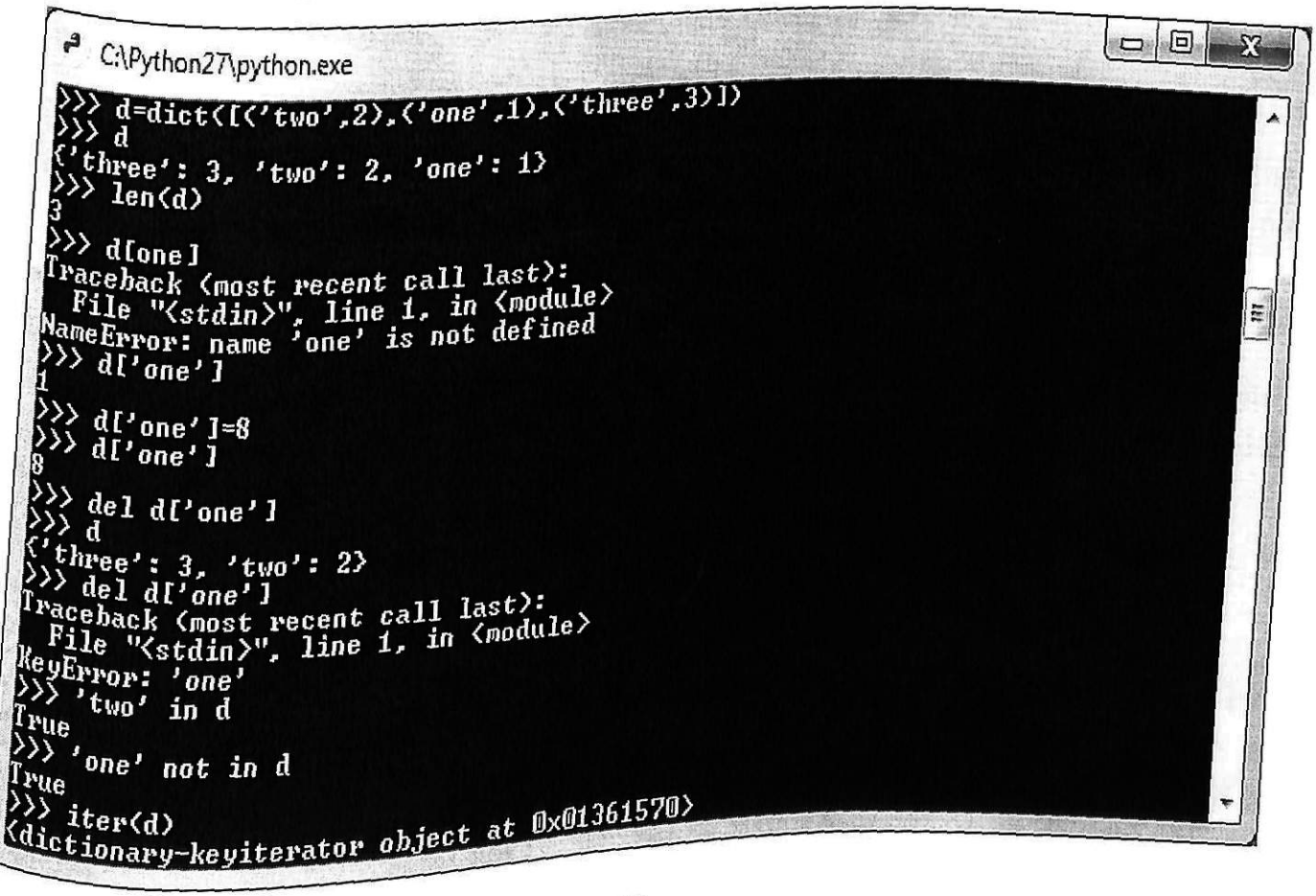
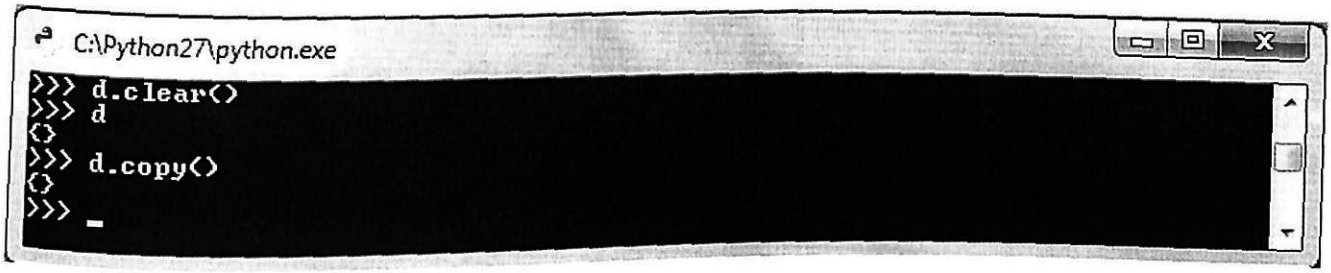


Figure B.2. – Operations that dictionaries support in Python.(cont.)



C. Ruby's Array class.

As in Ruby Array class has much more methods and to give the examples for all of them isn't the target for this work. So just the names and explanations of them were placed here in a table as a summary. And here also that class Array Mixes in Enumerable must be noted. That means the methods of Enumarable are also available in Array class.

Instance methods	Usage format	Explanation
&	enum & other_array → an_array	Set Intersection—Returns a new array containing elements common to the two arrays, with no duplicates.
*	enum * int → an_array enum * str → a_string	Repetition—With an argument that responds to to_str, equivalent to enum.join(str). Otherwise, returns a new array built by concatenating int copies of enum.
+	enum + other_array	Concatenation—Returns a new array built by concatenating the two arrays together to produce a

	→ an_array	third array.
-	enum - other_array → an_array	Array Difference—Returns a new array that is a copy of the original array, removing any items that also appear in other_array.
<<	enum << obj → enum	Append—Pushes the given object on to the end of this array. This expression returns the array itself, so several appends may be chained together.
<=>	enum <=> other_array → −1, 0, +1	Comparison—Returns an integer −1, 0, or +1 if this array is less than, equal to, or greater than other_array. Each object in each array is compared (using <=>). If any value isn't equal, then that inequality is the return value. If all the values found are equal, then the return is based on a comparison of the array lengths. Thus, two arrays are "equal" according to Array#<=> if and only if they have the same length and the value of each element is equal to the value of the corresponding element in the other array.
"=="	enum == obj → true or false	Equality—Two arrays are equal if they contain the same number of elements and if each element is equal to (according to Object#==) the corresponding element in the other array. If obj is not an array, attempt to convert it using to_ary and return obj==enum.
[]	enum[int] → obj or nil	Element Reference—Returns the element at index int, returns a subarray starting at index start and continuing for length elements, or returns a subarray specified by

	enum[start, length] → an_array or nil enum[range] → an_array or nil	range. Negative indices count backward from the end of the array (−1 is the last element). Returns nil if the index of the first element selected is greater than the array size. If the start index equals the array size and a length or range parameter is given, an empty array is returned. Equivalent to Array#slice.
[]=	enum[int] = obj → obj enum[start, length] = obj → obj enum[range] = obj → obj	Element Assignment—Sets the element at index int, replaces a subarray starting at index start and continuing for length elements, or replaces a subarray specified by range. If int is greater than the current capacity of the array, the array grows automatically. A negative int will count backward from the end of the array. Inserts elements if length is zero. If obj is an array, the form with the single index will insert that array into enum, and the forms with a length or with a range will replace the given elements in enum with the array contents. An IndexError is raised if a negative index points past the beginning of the array. (Prior to Ruby 1.9, assigning nil with the second and third forms of element assignment could delete the corresponding array elements; now it simply assigns nil to them.) See also Array#push and Array#unshift.
|	enum other_array → an_array	Set Union—Returns a new array by joining this array with other_array, removing duplicates. The rules for comparing elements are the same as for hash keys.

assoc	enum.assoc(obj) → an_array or nil	Searches through an array whose elements are also arrays comparing obj with the first element of each contained array using obj.== . Returns the first contained array that matches (that is, the first associated array) or nil if no match is found. See also Array#rassoc.
at	enum.at(int) → obj or nil	Returns the element at index int. A negative index counts from the end of enum. Returns nil if the index is out of range. See also Array#[].
clear	enum.clear → enum	Removes all elements from enum.
combination	enum.combination(size) → enumerator enum.combination(size) {|array| block } → enum	Constructs all combinations of the elements of enum of length size. If called with a block, passes each combination to that block; otherwise, returns an enumerator object. An empty result is generated if no combinations of the given length exist. See also Array#permutation.

As this list is going to be long enough, for a full view of Hashes' and Arrays' methods please take a look in this reference [2].

Additional-2

A. Python's *gc* — Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disablethe collector ifyou aresure your programdoes not createreference cycles. Automatic collectioncan be disabled by calling gc.disable(). To debug a leaking program call gc.set_debug(gc.DEBUG_LEAK).

Notice that this includes gc.DEBUG_SAVEALL, causing garbage-collected objects to be saved in gc.garbage for inspection. The gc module provides the following functions:

gc.enable() - Enable automatic garbage collection.

gc.disable() - Disable automatic garbage collection.

gc.isenabled() - Returns true if automatic collection is enabled.

gc.collect(generations=2) - With no arguments, run a full collection. The optional argument generation may be an integer specifying which generation to collect (from 0 to 2). A ValueError is raised if the generation number is invalid.

The number of unreachable objects found is returned.

The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular float.

gc.set_debug(flags) - Set the garbage collection debugging flags. Debugging information will be written to sys.stderr. See below for a list of debugging flags which can be combined using bit operations to control debugging.

gc.get_debug() - Return the debugging flags currently set.

gc.get_objects() - Returns a list of all objects tracked by the collector, excluding the list returned.

gc.set_threshold(threshold0[, threshold1[, threshold2]]) - Set the garbage collection thresholds (the collection frequency). Setting threshold0 to zero disables collection.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation 0). If an object survives a collection it is moved into the next older generation. Since generation 2 is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds threshold0, collection starts. Initially only generation 0 is examined. If generation 0 has been examined more than threshold1 times since generation 1 has been examined, then generation 1 is examined as well. Similarly, threshold2 controls the number of collections of generation 1 before collecting generation 2.

gc.get_count() - Return the current collection counts as a tuple of (count0, count1, count2).

gc.get_threshold() - Return the current collection thresholds as a tuple of (threshold0, threshold1, threshold2).

gc.get_referrers(*objs) - Return the list of objects that directly refer to any of objs. This function will only locate those containers which support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.

Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call collect() before calling get_referrers().

Care must be taken when using objects returned by get_referrers() because some of them could still be under construction and hence in a temporarily invalid state. Avoid using get_referrers() for any purpose other than debugging.

gc.get_referents(*objs) - Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments' C-level tp_traverse methods (if any), and may not be all objects actually directly reachable. tp_traverse methods are supported only by objects that support garbage collection,

and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

gc.is_tracked(obj) - Returns True if the object is currently tracked by the garbage collector, False otherwise. As a general rule, instances of atomic types aren't tracked and instances of non-atomic types (containers, user-defined objects...) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances (e.g. dicts containing only atomic keys and values).

The following variables are provided for read-only access (you can mutate the values but should not rebind them):

gc.garbage - A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects).

By default, this list contains only objects with __del__() methods. Objects that have __del__() methods and are part of a reference cycle cause the entire reference cycle to be uncollectable, including objects not necessarily in the cycle but reachable only from it. Python doesn't collect such cycles automatically because, in general, it isn't possible for Python to guess a safe order in which to run the __del__() methods. If you know a safe order, you can force the issue by examining the garbage list, and explicitly breaking cycles due to your objects within the list. Note that these objects are kept alive even so by virtue of being in the garbage list, so they should be removed from garbage too. For example, after breaking cycles, do del gc.garbage[:] to empty the list. It's generally better to avoid the issue by not creating cycles containing objects with __del__() methods, and garbage can be examined in that case to verify that no such cycles are being created.

If DEBUG_SAVEALL is set, then all unreachable objects will be added to this list rather than freed. Changed in version 3.2: If this list is non-empty at interpreter shutdown, a ResourceWarning is emitted, which is silent by default. If DEBUG_UNCOLLECTABLE is set, in addition all uncollectable objects are printed.

gc.callbacks - A list of callbacks that will be invoked by the garbage collector before and after collection. The callbacks will be called with two arguments, phase and info.

phase can be one of two values:

"start": The garbage collection is about to start.

"stop": The garbage collection has finished.

info is a dict providing more information for the callback. The following keys are currently defined:

"generation": The oldest generation being collected.

"collected": When phase is "stop", the number of objects successfully collected.

"uncollectable": When phase is "stop", the number of objects that could not be collected and were put in garbage.

Applications can add their own callbacks to this list. The primary use cases are:

Gathering statistics about garbage collection, such a show often various generations are collected, and how long the collection takes.

Allowing applications to identify and clear their own uncollectable types when they appear in garbage.

The following constants are provided for use with set_debug():

gc.DEBUG_STATS - Print statistics during collection. This information can be useful when tuning the collection frequency.

gc.DEBUG_COLLECTABLE - Print information on collectable objects found.

gc.DEBUG_UNCOLLECTABLE - Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the garbage list.

gc.DEBUG_SAVEALL - When set, all unreachable objects found will be appended to garbage rather than being freed. This can be useful for debugging a leaking program.

gc.DEBUG_LEAK - The debugging flags necessary for the collector to print information about a leaking program (equal to DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_SAVEALL).

B. Ruby's GC

The GC module provides an interface to Ruby's mark and sweep garbage collection mechanism. Some of the underlying methods are also available via the ObjectSpace module.

Module methods	Usage format	Explanation
count	GC.count → int	Returns a count of the number of times GC has run in the current process.
disable	GC.disable → true or false	Disables garbage collection, returning true if garbage collection was already disabled.
enable	GC.enable → true or false	Enables garbage collection, returning true if garbage collection was disabled.
start	GC.start → nil	Initiates garbage collection, unless manually disabled.
stress	GC.stress → true or false	Returns the current value of the stress flag (see GC.stress=).
stress=	GC.stress = bool → bool	Ruby will normally run garbage collection periodically. Setting the stress flag to true forces garbage collection to occur every time Ruby allocates

		a new object. This is typically used only for testing extensions (and Ruby itself).
garbage_collect	garbage_collect → nil	(Instance method) Equivalent to GC.start.