SAGE For Newbies

by Ted Kosan

Table of Contents

1 Preface

1.1 Dedication

This book is dedicated to Steve Yegge and his blog entry "Math Every Day" (http://steve.yegge.googlepages.com/math-every-day).

1.1 Acknowledgments

The following people have provided feedback on this book (if I forgot to include your name on this list, please email me at ted.kosan at gmail.com):

Dave Dobbs

David Joyner

Greg Landweber

Jeremy Pedersen

William Stein

Steve Vonn

Joe Wetherell

1.2 Support Group

The support group for this book is called **sage-support** and it can be accessed at:

http://groups.google.com/group/sage-support . Please place "[Newbies book]" in the title of your email when you post to this group.

2 Introduction

SAGE is an open source mathematics computing environment (MCE) for performing symbolic, algebraic, and numerical computations. Mathematics computing environments are complex and it takes a significant amount of time and effort to become proficient at using one. The amount of power that a mathematics computing environment makes available to a user, however, is well worth the effort needed to learn one. It will take a beginner a while to become an expert at using SAGE, but fortunately one does not need to be a SAGE expert in order to begin using it to solve problems.

2.1 What Is A Mathematics Computing Environment?

A mathematics computing environment is a set of computer programs that are able to automatically perform a wide range of mathematics calculation algorithms. Calculation algorithms exist for almost all areas of mathematics and new algorithms are being developed all the time.

A significant number of mathematics computing environments have been created since the 1960s and the following list contains some of the more popular ones:

http://en.wikipedia.org/wiki/Comparison_of_computer_algebra_systems

Some environments are highly specialized and some are general purpose. Some allow mathematics to be displayed and entered in traditional form (which is what is found in most math textbooks), some are able to display traditional form mathematics but need to have it input as text, and some are only able to have mathematics displayed and entered as text.

As an example of the difference between traditional mathematics form and text form, here is a formula which is displayed in traditional form:

$$A = x^2 + 4 \cdot h \cdot x$$

and here is the same formula in text form:

```
A == x^2 + 4*h*x
```

Most mathematics computing environments contain some kind of mathematics-oriented high-level programming language. This allows software programs to be developed which have access to the mathematics algorithms which are included in the environment. Some of these mathematics-oriented programming languages were created specifically for the environment they work in while others are built around an existing programming language.

Some mathematics computing environments are proprietary and need to be purchased while others are open source and available for free. Both kinds of environments possess similar core capabilities, but they usually differ in other areas.

Proprietary environments tend to be more polished than open source environments and they often have graphical user interfaces that make inputting and manipulating mathematics in traditional form relatively easy. However, proprietary environments also have drawbacks. One drawback is that there is always a chance that the company that owns it may go out of business and this may make the environment unavailable for further use. Another drawback is that users are unable to enhance a proprietary environment because the environment's source code is not made available to users.

Open source mathematics computing environments usually do not have graphical user interfaces, but their user interfaces are adequate for most purposes and the environment's source code will always be available to whomever wants it. This means that people can use the environment for as long as there is interest in it and they can also enhance it as desired.

2.2 What Is SAGE?

SAGE is an open source mathematics computing environment that inputs mathematics in textual form and displays it in either textual form or traditional form. While most mathematics computing environments are self-contained entities, SAGE takes the umbrella-like approach of providing some algorithms itself and some by wrapping around other mathematics computing environments. This strategy allows SAGE to provide the power of multiple mathematics computing environments within an architecture that is easily able to evolve to meet future needs.

SAGE is written in the powerful and very popular Python programming language and the mathematics-oriented programming language that SAGE makes available to users is an extension of Python. This means that expert SAGE users must also be expert Python programmers. Some knowledge of the Python programming language is so critical to being able to successfully use SAGE that a user's knowledge of Python can be used to help determine their level of SAGE expertise. (see Table 1)

Level	Knowledge
SAGE Expert	Knows Python well and SAGE well.
SAGE Novice	Knows Python but has only used SAGE for a short while.
SAGE Newbie	Does not know Python but has been exposed to at least 1 programming language.
Programming Newbie	Does not know how a computer works and has never programmed before.

Table 1: SAGE user experience levels

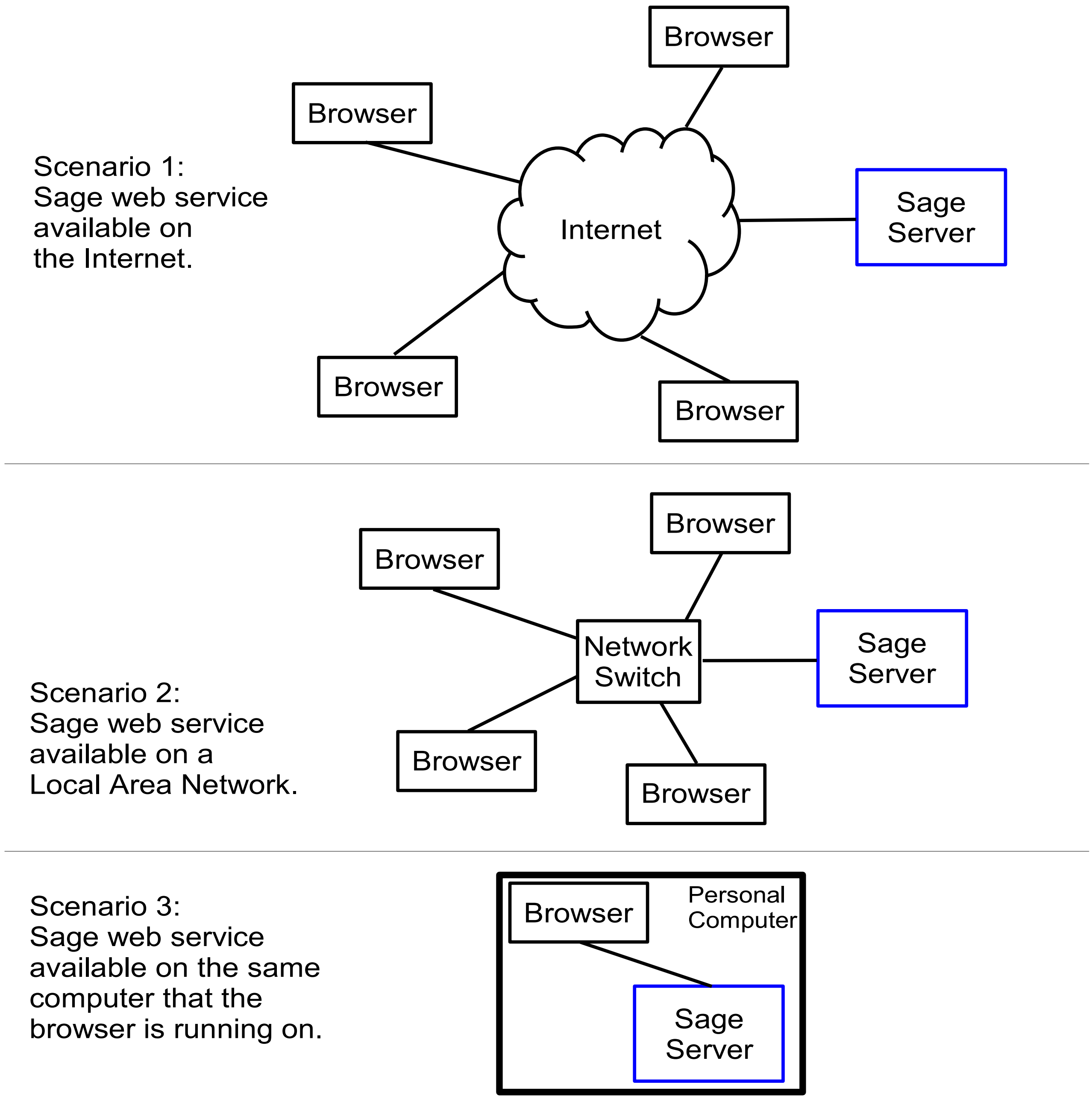
This book is for SAGE Newbies. It assumes the reader has been exposed to at least 1 programming language, but has never programmed in Python (if your understanding of how computer programming works needs refreshing, you may want to read through the Fundamentals Of Computing section of this book.) This book will teach you enough Python to begin solving problems with SAGE. It will help you to become a SAGE Novice, but you will need to learn Python from books that are dedicated to it before you can become a SAGE Expert.

If you are a programming newbie, this book will probably be too advanced for you. I have written a series of free books called *The Professor and Pat Programming Series* (http://professorandpat.org) and they are designed for programming newbies. If you are a programming newbie and are interested in learning how to use SAGE, you might be interested in working through the Professor and Pat Programming books first and then come back to this book when you are finished with them.

The SAGE website (sagemath.org) contains more information about SAGE along with other SAGE resources.

2.3 Accessing SAGE As A Web Service

The ways in which SAGE can be used are as flexible as its architecture. Most SAGE beginners, however, will first use SAGE as a web service which is accessed using a web browser. Any copy of SAGE can be configured to provide this web service. Drawing 2.1 shows 3 SAGE web service scenarios:



Drawing 2.1: Three SAGE web service scenarios.

2.3.1 Accessing SAGE As A Web Service Using Scenario 1

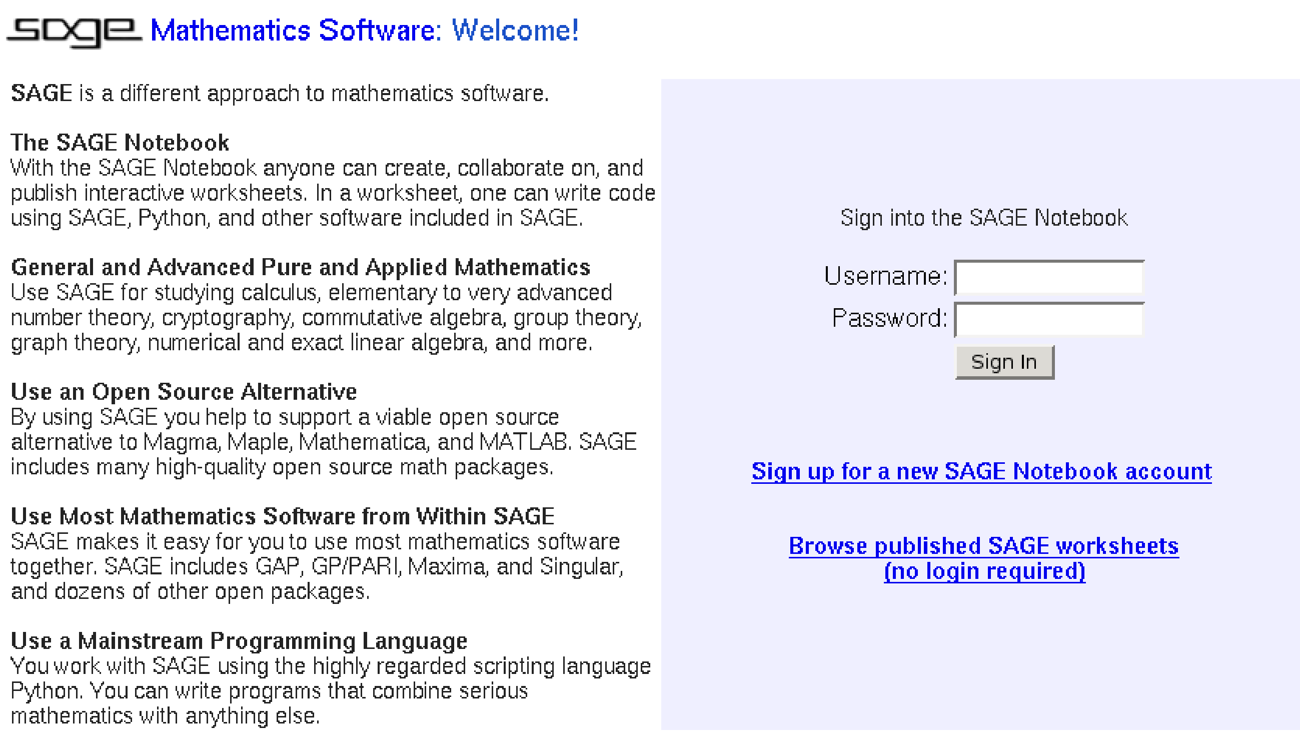
SAGE currently works best with the Firefox web browser and if you do not yet have Firefox installed on your computer, it can be obtained at http://mozilla.com/firefox.

The SAGE development team provides a public SAGE web service at (http://sagenb.com) and this service can also be accessed from the top of the SAGE homepage. We will now walk through the steps that are needed to sign up for an account on this public SAGE web service.

Open a Firefox browser window and enter the following into the URL bar:

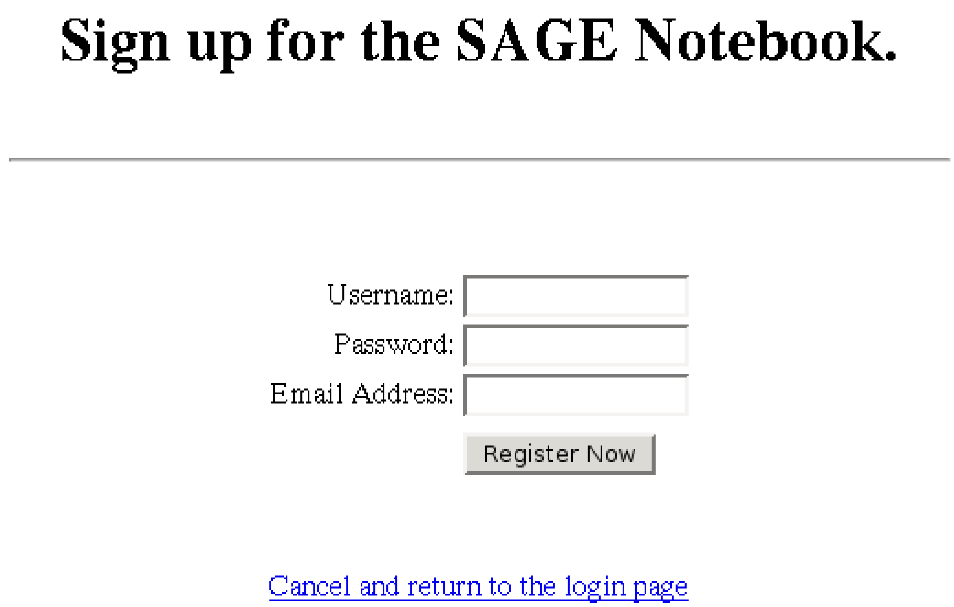
```
http://sagenb.com
```

The service will then display a Welcome page (see Drawing 2.2)



Drawing 2.2: SAGE Welcome screen.

The SAGE web service is called a SAGE **Notebook** because it simulates the kind of notebook that mathematicians traditionally use to perform mathematical calculations. Before you can access the Notebook, you must first sign up for a Notebook account. Select the **Sign up for a new SAGE Notebook account** link and a registration page will be displayed. (see Drawing 2.3)

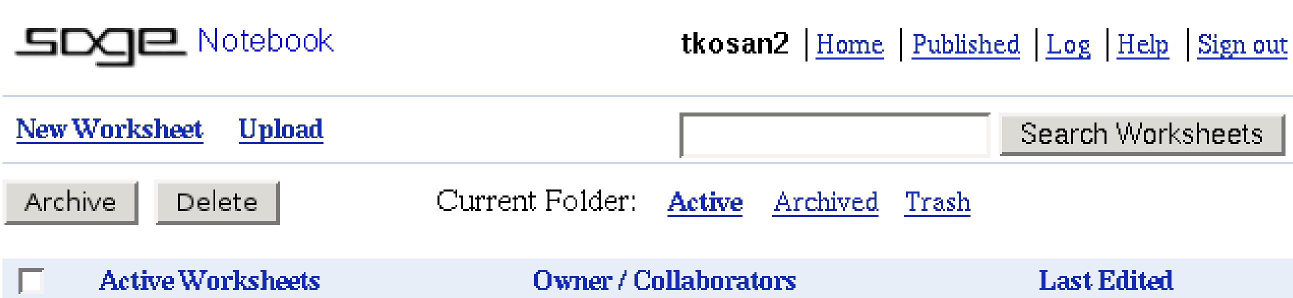


Drawing 2.3: Signup page.

Enter a username and password in the Username and Password text boxes and then press the **Register Now** button. A page will then be displayed that indicates that the registration information was received and that a confirmation message was sent to the email address that you provided.

Open this email and select the link that it contains. This will complete the registration process and then you may go back to the Notebook's **Welcome** page and log in.

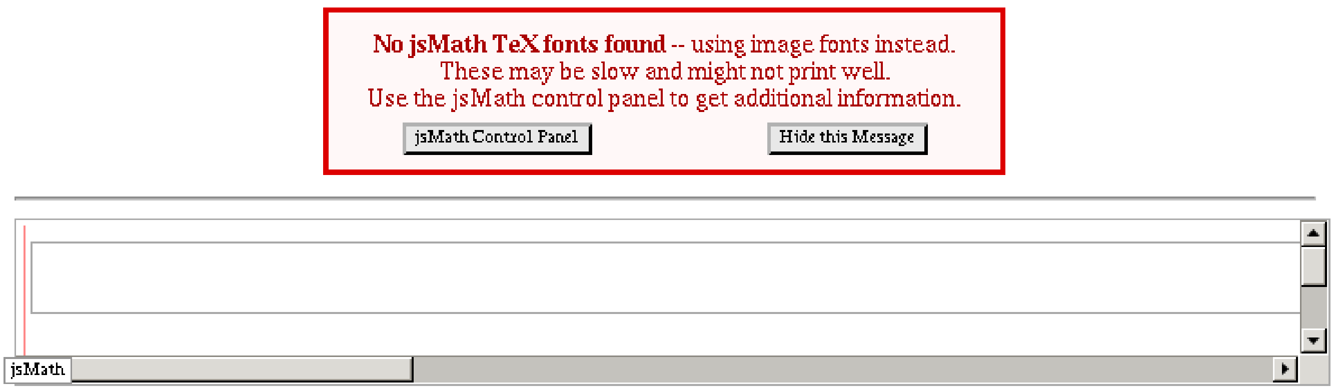
After successfully logging into your Notebook account, a **worksheet management** page will be displayed. (see Drawing 2.4)



Drawing 2.4: Worksheet management page.

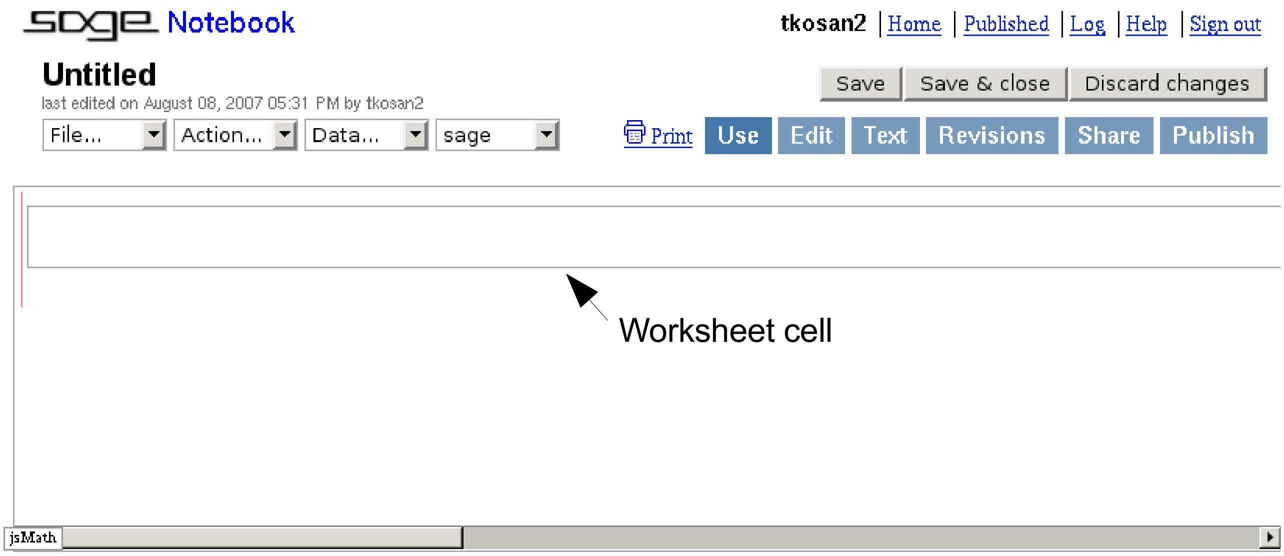
Physical mathematics notebooks contain worksheets and therefore SAGE's virtual notebook contains worksheets too. The worksheet management page allows worksheets to be created, deleted, published on the Internet, etc. Since this is a newly created Notebook, it does not contain any worksheets yet.

Create a new worksheet now by selecting the **New Worksheet** link. A worksheet can either use special mathematics fonts to display mathematics in traditional form or it can use images of these fonts. If the computer you are working on does not have mathematics fonts installed, the worksheet will display a message which indicates that it will use its built-in image fonts as an alternative. (see Drawing 2.5)



Drawing 2.5: jsMath No TeXfonts alert.

The image fonts are not as clear as normal mathematics fonts, but they are adequate for most purposes. Later you can install mathematics fonts on your computer if you would like, but for now just press the **Hide this Message** button and a page which contains a blank worksheet will be shown. (see Drawing 2.6)



Drawing 2.6: Blank worksheet.

Worksheets contain 1 or more **cells** which are used to enter source code that will be executed by SAGE. Cells have rectangles drawn around them as shown in Figure 6 and they are able to grow larger as more text is entered into them. When a worksheet is first created, an initial cell is placed at the top of its work area and this is where you will normally begin entering text.

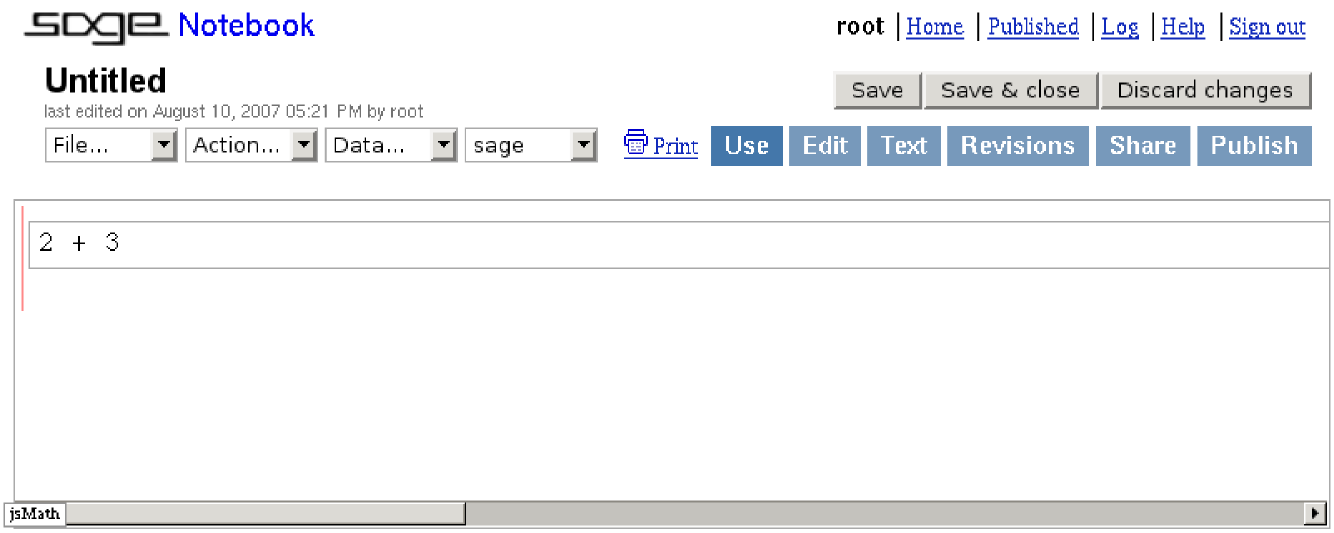
2.4 Entering Source Code Into A SAGE Cell

Lets begin exploring SAGE by using it as a simple calculator. Place your mouse cursor inside of the cell that is at the top of your worksheet. Notice that the cursor is automatically placed against the left side of a new cell. You must always begin each line of SAGE source code at the left side of a cell with no indenting (unless you are instructed to do otherwise).

Type the following text, but do not press the enter key:

```
2 + 3
```

your worksheet should now look like Drawing 2.7.



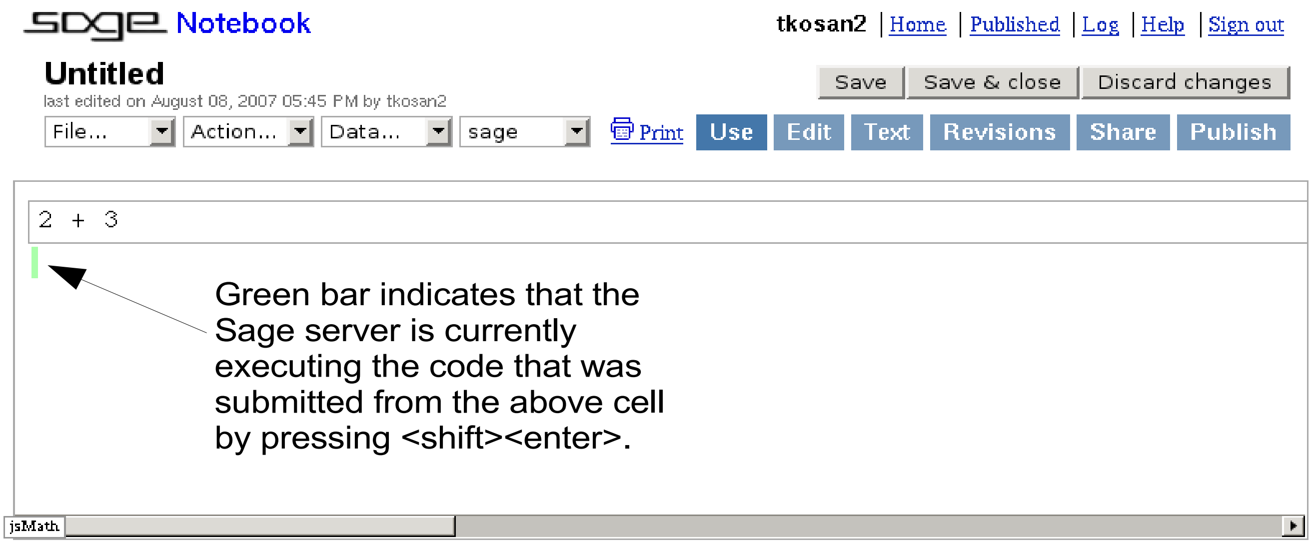
Drawing 2.7: Entering text into a cell.

At this point you have 2 choices. You can either press the **enter key** <enter> or you can **hold down the shift key and press the enter key** <shift><enter>. If you simply press the enter key, the cell will expand and drop the cursor down to the next line so you can continue entering source code.

If you press **shift** and **enter**, however, the Worksheet will take all the source code that has been typed into the cell and send it to the SAGE server through the network so the server can **execute** the code. When SAGE is given source code to execute, it will first process it using software called the **SAGE preprocessor**. The preprocessor converts SAGE source code into Python source code so that it can be executed using the Python environment that SAGE is built upon.

The converted source code is then passed to the Python environment where it is compiled into a special form of machine language called **Python bytecode**. The bytecode is then executed by a program that emulates a hardware CPU and this program is called the **Python interpreter**.

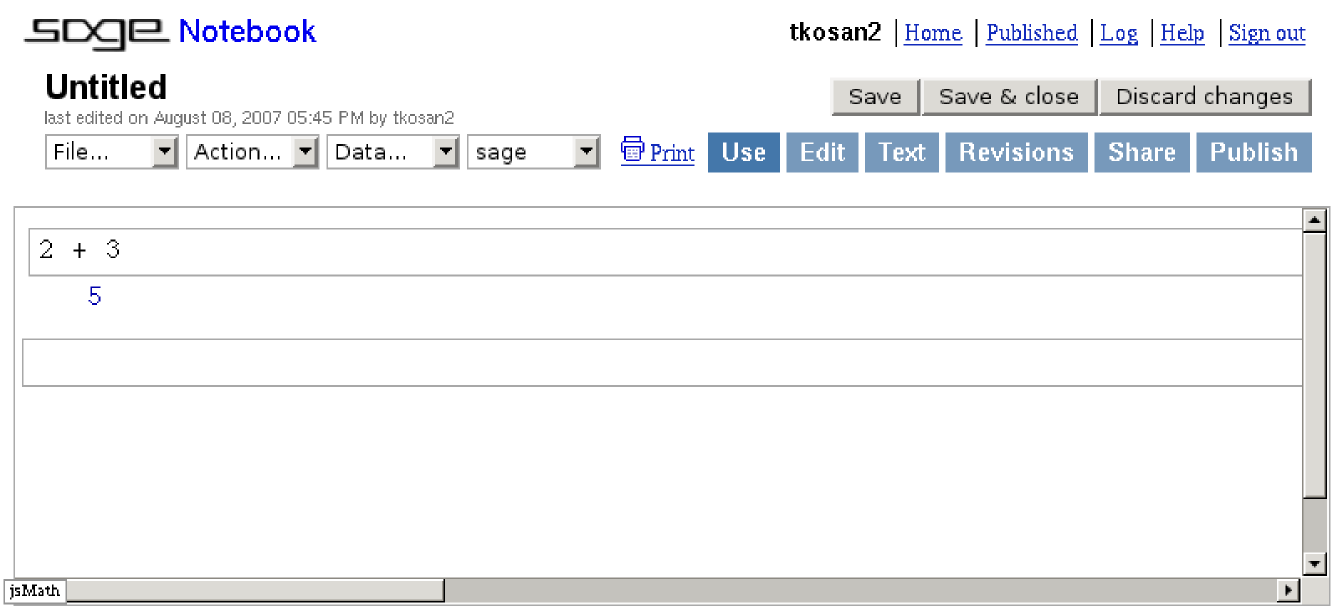
Sometimes the server is able to execute the code quickly and sometimes it will take a while. While the code is being executed by the server, the Worksheet will display a small green vertical bar beneath the cell towards the left side of the window as shown in Drawing 2.8.



Drawing 2.8: Executing the text in a cell.

When the server is finished executing the source code, the green bar will disappear. If a displayable result was generated, this result is sent back to the Worksheet and the Worksheet then displays it in the area that is directly beneath the cell that the request was submitted from.

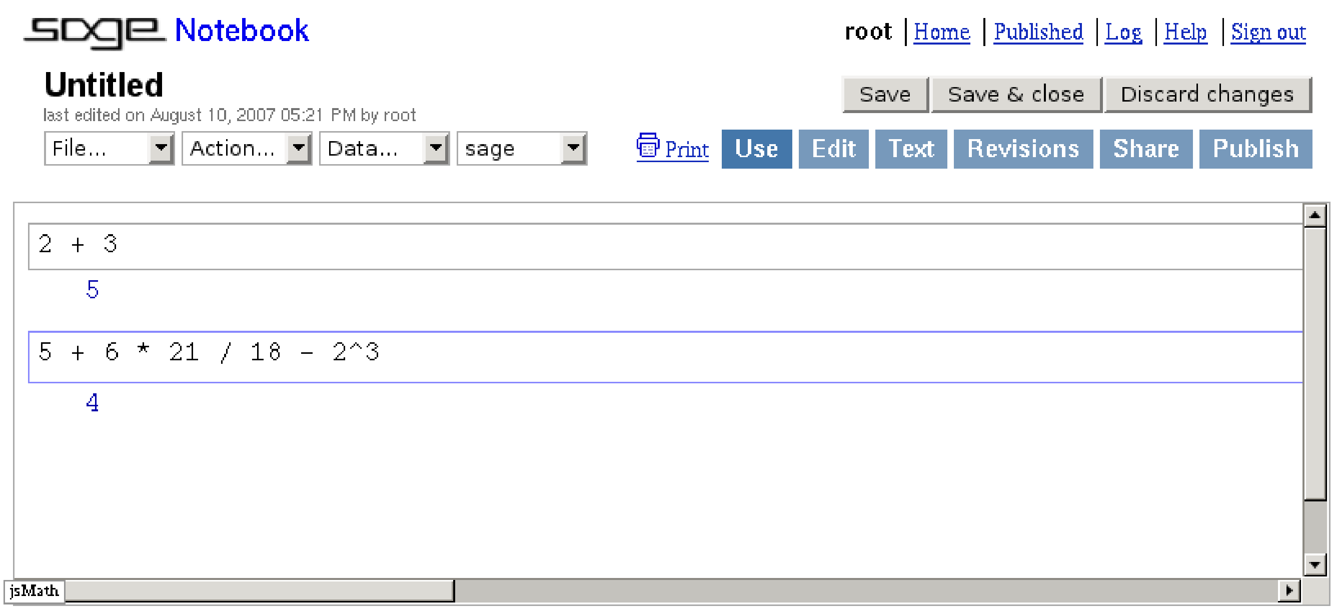
Press **shift** and **enter** in your cell now and in a few moments you should see a result that looks like Drawing 2.9.



Drawing 2.9: The results of execution are displayed.

If code was submitted for execution from the bottom cell in the Notebook, a blank cell is automatically added beneath this cell when the server has finished executing the code.

Now enter the source code that is shown in the second cell in Drawing 2.10 and execute it.



Drawing 2.10: A more complex calculation

3 SAGE Programming Fundamentals

3.1 Objects, Values, And Expressions

The source code lines

```
2 + 3
```

and

```
5 + 6*21/18 - 2^3
```

are both called **expressions** and the following is a definition of what an expression is:

> An **expression** in a programming language is a combination of values, variables, operators, and functions that are interpreted (*evaluated*) according to the particular rules of precedence and of association for a particular programming language, which computes and then produces another value. The expression is said to *evaluate to* that value. As in mathematics, the expression *is* (or can be said to *have*) its evaluated value; the expression is a representation of that value. (http://en.wikipedia.org/wiki/Expression_(programming))

In a computer, a **value** is a pattern of bits in one or more memory locations that mean something when interpreted using a given context. In SAGE, patterns of bits in memory that have meaning are called **objects**. SAGE itself is built with objects and the data that SAGE programs process are also represented as objects. Objects are explained in more depth in Chapter 4.

In the above expressions, 2, 3, 5, 6, 21, and 18 are objects that are interpreted using a context called the **sage.rings.integer.Integer** context. Contexts that can be associated with objects are called **types** and an object that is of type sage.rings.integer.Integer is used to represent integers.

There is a command in SAGE called **type()** which will return the **type** of any object that is passed to it. Lets have the type() command tell us what the type of the objects 3 and 21 are by executing the following code: (Note: from this point forward, the source code that is to be entered into a cell, and any results that need to be displayed, will be given without using a graphic worksheet screen capture.)

```
type(3)
|
    <type 'sage.rings.integer.Integer'>
```

```
type(21)
|
    <type 'sage.rings.integer.Integer'>
```

The way that a person tells the type() command what object they want to see the type information for is by placing the object within the parentheses which are to the right of the the name 'type'.

3.2 Operators

In the above expressions, the characters +, −, *, /, ^ are called **operators** and their purpose is to tell SAGE what operations to perform on the objects in an expression. For example, in the expression 2 + 3, the **addition** operator + tells SAGE to add the integer 2 to the integer 3 and return the result. Since both the objects 2 and 3 are of type sage.rings.integer.Integer, the result that is obtained by adding them together will also be an object of type sage.rings.integer.Integer.

The **subtraction** operator is −, the **multiplication** operator is *, / is the **division** operator, % is the **remainder** operator, and ^ is the **exponent** operator. SAGE has more operators in addition to these and more information about them can be found in Python documentation.

The following examples show the −, *, /,%, and ^ operators being used:

```
5 - 2
|
   3
```

```
3*4
|
   12
```

```
30/3
|
   10
```

```
8%5
|
   3
```

```
2^3
|
   8
```

The − character can also be used to indicate a negative number:

```
-3
```

```
    -3
```

Subtracting a negative number results in a positive number:

```
- -3
```

```
    3
```

3.3 Operator Precedence

When expressions contain more than 1 operator, SAGE uses a set of rules called **operator precedence** to determine the order in which the operators are applied to the objects in the expression. Operator precedence is also referred to as the **order of operations**. Operators with higher precedence are evaluated before operators with lower precedence. The following table shows a subset of SAGE's operator precedence rules with higher precedence operators being placed higher in the table:

^	Exponents are evaluated right to left.
*,%,/	Then multiplication, remainder, and division operations are evaluated left to right.
+, −	Finally, addition and subtraction are evaluated left to right.

Lets manually apply these precedence rules to the multi-operator expression we used earlier. Here is the expression in source code form:

```
5 + 6*21/18 - 2^3
```

And here it is in traditional form:

$$5 + \frac{6 \cdot 21}{18} - 2^3$$

According to the precedence rules, this is the order in which SAGE evaluates the operations in this expression:

```
5 + 6*21/18 - 2^3
5 + 6*21/18 - 8
5 + 126/18 - 8
5 + 7 - 8
12 - 8
4
```

Starting with the first expression, SAGE evaluates the ^ operator first which results in the 8 in the expression below it. In the second expression, the * operator is executed next, and so on. The last expression shows that the final result after all of the operators have been evaluated is 4.

3.4 Changing The Order Of Operations In An Expression

The default order of operations for an expression can be changed by grouping various parts of the expression within parentheses. Parentheses force the code that is placed inside of them to be evaluated before any other operators are evaluated. For example, the expression 2 + 4*5 evaluates to 22 using the default precedence rules:

```
2 + 4*5
|
    22
```

If parentheses are placed around 4 + 5, however, the addition is forced to be evaluated before the multiplication and the result is 30:

```
(2 + 4)*5
|
    30
```

Parentheses can also be nested and nested parentheses are evaluated from the most deeply nested parentheses outward:

```
((2 + 4)*3)*5
|
    90
```

Since parentheses are evaluated before any other operators, they are placed at the top of the precedence table:

() Parentheses are evaluated from the inside out.

^ Then exponents are evaluated right to left.

*,%,/ Then multiplication, remainder, and division operations are evaluated left to right.

+, − Finally, addition and subtraction are evaluated left to right.

3.5 Variables

A variable is a **name** that can be associated with a memory address so that humans can refer to bit pattern symbols in memory using a **name** instead of a **number**. One way to create variables in SAGE is through **assignment** and it consists of placing the name of a variable you would like to create on the left side of an equals sign '=' and an expression on the right side of the equals sign. When the expression returns an object, the object is assigned to the variable.

In the following example, a variable called **box** is created and the number 7 is assigned to it:

```
box = 7
```

Notice that unlike earlier examples, a displayable result is not returned to the worksheet because the result was placed in the variable **box**. If you want to see the contents of box, type its name into a blank cell and then evaluate the cell:

```
box

    7
```

As can be seen in this example, variables that are created in a given cell in a worksheet are also available to the other cells in a worksheet. Variables exist in a worksheet as long as the worksheet is open, but when the worksheet is closed, the variables are lost. When the worksheet is reopened, the variables will need to be created again by evaluating the cells they are assigned in. Variables can be saved before a worksheet is closed and then loaded when the worksheet is opened again, but this is an advanced topic which will be covered later.

SAGE variables are also case sensitive. This means that SAGE takes into account the case of each letter in a variable name when it is deciding if two or more variable names are the same variable or not. For example, the variable name **Box** and the variable name **box** are not the same variable because the first variable name starts with an upper case 'B' and the second variable name starts with a lower case 'b'.

Programs are able to have more than 1 variable and here is a more sophisticated example which uses 3 variables:

```
a = 2
```

```
b = 3
```

```
a + b

    5
```

```
answer = a + b
```

```
answer
    5
```

The part of an expression that is on the right side of an equals sign '=' is always evaluated first and the result is then assigned to the variable that is on the left side of the equals sign.

When a variable is passed to the type() command, the type of the object that the variable is assigned to is returned:

```
a = 4
type(a)
    <type 'sage.rings.integer.Integer'>
```

Data types and the type command will be covered more fully later.

3.6 Statements

Statements are the part of a programming language that is used to encode algorithm logic. Unlike expressions, statements do not return objects and they are used because of the various effects they are able to produce. Statements can contain both expressions and statements and programs are constructed by using a sequence of statements.

3.6.1 The print Statement

If more than one expression in a cell generates a displayable result, the cell will only display the result from the bottommost expression. For example, this program creates 3 variables and then attempts to display the contents of these variables:

```
a = 1
b = 2
c = 3
a
b
c
    3
```

In SAGE, programs are executed one line at a time, starting at the topmost line of code and working downwards from there. In this example, the line a = 1 is executed first, then the line b = 2 is executed, and so on. Notice, however, that even though we wanted to see what was in all 3 variables, only the content of the last variable was displayed.

SAGE has a statement called **print** that allows the results of expressions to be displayed regardless of where they are located in the cell. This example is similar to the previous one except print statements are used to display the contents of all 3 variables:

```
a = 1
b = 2
c = 3
print a
print b
print c

    1
    2
    3
```

The print statement will also print multiple results on the same line if commas are placed between the expressions that are passed to it:

```
a = 1
b = 2
c = 3*6
print a,b,c

   1 2 18
```

When a comma is placed after a variable or object which is being passed to the print statement, it tells the statement not to drop the cursor down to the next line after it is finished printing. Therefore, the next time a print statement is executed, it will place its output on the same line as the previous print statement's output.

Another way to display multiple results from a cell is by using semicolons ';'. In SAGE, semicolons can be placed after statements as optional terminators, but most of the time one will only see them used to place multiple statements on the same line. The following example shows semicolons being used to allow variables a, b, and c to be initialized on one line:

```
a=1;b=2;c=3
print a,b,c

   1 2 3
```

The next example shows how semicolons can be also used to output multiple results from a cell:

```
a = 1
b = 2
c = 3*6
a;b;c
|
    1
    2
    18
```

3.7 Strings

A **string** is a type of object that is used to hold **text-based** information. The typical expression that is used to create a string object consists of text which is enclosed within either **double quotes** or **single quotes**. Strings can be referenced by variables just like numbers can and strings can also be displayed by the print statement. The following example assigns a string object to the variable 'a', prints the string object that 'a' references, and then also displays its type:

```
a = "Hello, I am a string."
print a
type(a)
|
    Hello, I am a string.
    <type 'str'>
```

3.8 Comments

Source code can often be difficult to understand and therefore all programming languages provide the ability for comments to be included in the code. Comments are used to explain what the code near them is doing and they are usually meant to be read by a human looking at the source code. Comments are ignored when the program is executed.

There are two ways that SAGE allows comments to be added to source code. The first way is by placing a **pound sign** '#' to the left of any text that is meant to serve as a comment. The text from the pound sign to the end of the line the pound sign is on will be treated as a comment. Here is a program that contains comments which use a pound sign:

```
#This is a comment.
x = 2 #Set the variable x equal to 2.
print x
```

```
    2
```

When this program is executed, the text that starts with a pound sign is ignored.

The second way to add comments to a SAGE program is by enclosing the comments in a set of triple quotes. This option is useful when a comment is too large to fit on one line. This program shows a triple quoted comment:

```
"""
This is a longer comment and it uses
more than one line. The following
code assigns the number 3 to variable
x and then it prints x.
"""

x = 3
print x
```

```
    3
```

3.9 Conditional Operators

A **conditional operator** is an operator that is used to compare two objects. Expressions that contain conditional operators return a **boolean** object and a **boolean** object is one that can either be **True** or **False**. Table 2 shows the conditional operators that SAGE uses:

Operator	Description
x == y	Returns **True** if the two objects are equal and **False** if they are not equal. Notice that == performs a comparison and not an assignment like = does.
x <> y	Returns **True** if the objects are not equal and **False** if they are equal.
x != y	Returns **True** if the objects are not equal and **False** if they are equal.
x < y	Returns **True** if the left object is less than the right object and **False** if the left object is not less than the right object.
x <= y	Returns **True** if the left object is less than or equal to the right object and **False** if the left object is not less than or equal to the right object.
x > y	Returns **True** if the left object is greater than the right object and **False** if the left object is not greater than the right object.
x >= y	Returns **True** if the left object is greater than or equal to the right object and **False** if the left object is not greater than or equal to the right object.

Table 2: Conditional Operators

The following examples show each of the conditional operators in Table 2 being used to compare objects that have been placed into variables x and y:

```
# Example 1.
x = 2
y = 3

print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
|
    2 == 3 : False
    2 <> 3 : True
    2 != 3 : True
    2 < 3 : True
    2 <= 3 : True
    2 > 3 : False
    2 >= 3 : False
```

```
# Example 2.
x = 2
y = 2

print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
|
    2 == 2 : True
    2 <> 2 : False
    2 != 2 : False
    2 < 2 : False
    2 <= 2 : True
    2 > 2 : False
    2 >= 2 : True
```

```
# Example 3.
x = 3
y = 2
```

```
print x, "==", y, ":", x == y
print x, "<>", y, ":", x <> y
print x, "!=", y, ":", x != y
print x, "<", y, ":", x < y
print x, "<=", y, ":", x <= y
print x, ">", y, ":", x > y
print x, ">=", y, ":", x >= y
|
    3 == 2 : False
    3 <> 2 : True
    3 != 2 : True
    3 < 2 : False
    3 <= 2 : False
    3 > 2 : True
    3 >= 2 : True
```

Conditional operators are placed at a lower level of precedence than the other operators we have covered to this point:

() Parentheses are evaluated from the inside out.

^ Then exponents are evaluated right to left.

*,%,/ Then multiplication, remainder, and division operations are evaluated left to right.

+, − Then addition and subtraction are evaluated left to right.

==,<>,!=,<,<=,>,>= Finally, conditional operators are evaluated.

3.10 Making Decisions With The if Statement

All programming languages provide the ability to make decisions and the most commonly used statement for making decisions in SAGE is the **if** statement.

A simplified syntax specification for the **if** statement is as follows:

```
if <expression>:
   <statement>
   <statement>
   <statement>
   .
   .
   .
```

The way an **if** statement works is that it evaluates the **expression** to its immediate right and then looks at the **object** that is returned. If this object is "true", the statements that are **inside** the **if** statement are executed. If the object is "false", the statements inside of the **if** are not executed.

In SAGE, an object is "true" if it is **nonzero** or **nonempty** and it is "false" if it is **zero** or **empty**. An expression that contains one or more conditional operators will return a **boolean** object which will be either **True** or **False**.

The way that statements are placed inside of a statement is by putting a **colon** ':' at the end of the statement's header and then placing one or more statements underneath it. The statements that are placed underneath an enclosing statement must each be indented one or more **tabs** or **spaces** from the left side of the enclosing statement. All indented statements, however, must be indented the same way and the same amount. One or more statements that are indented like this are referred to as a **block** of code.

The following program uses an **if** statement to determine if the number in variable x is greater than 5. If x is greater than 5, the program will print "Greater" and then "End of program".

```
x = 6

print x > 5

if x > 5:
    print x
    print "Greater"

print "End of program"

    True
    6
    Greater
    End of program
```

In this program, x has been set to 6 and therefore the expression x > 5 is true. When this expression is printed, it prints the boolean object **True** because 6 is greater than 5.

When the **if** statement evaluates the expression and determines it is **True**, it then executes the print statements that are inside of it and the contents of variable x are printed along with the string "Greater". If additional statements needed to be placed within the **if** statement, they would have been added underneath the print statements at the same level of indenting.

Finally, the last print statement prints the string "End of program" regardless of what the **if** statement does.

Here is the same program except that x has been set to 4 instead of 6:

```
x = 4

print x > 5

if x > 5:
    print x
    print "Greater."

print "End of program."

    False
    End of program.
```

This time the expression x > 4 returns a **False** object which causes the **if** statement to not execute the statements that are inside of it.

3.11 The and, or, And not Boolean Operators

Sometimes one wants to check if two or more expressions are all true and the way to do this is with the **and** operator:

```
a = 7
b = 9
print a < 5 and b < 10
print a > 5 and b > 10
print a < 5 and b > 10
print a > 5 and b < 10
if a > 5 and b < 10:
    print "These expressions are both true."

   False
   False
   False
   True
   These expressions are both true.
```

At other times one wants to determine if at least one expression in a group is true and this is done with the **or** operator:

```
a = 7
b = 9
print a < 5 or b < 10
```

```
print a > 5 or b > 10
print a > 5 or b < 10
print a < 5 or b > 10

if a < 5 or b < 10:
    print "At least one of these expressions is true."
```

```
True
True
True
False
At least one of these expressions is true.
```

Finally, the **not** operator can be used to change a True result to a False result, and a False result to a True result:

```
a = 7
print a > 5
print not a > 5
```

```
True
False
```

Boolean operators are placed at a lower level of precedence than the other operators we have covered to this point:

() Parentheses are evaluated from the inside out.

^ Then exponents are evaluated right to left.

*,%,/ Then multiplication, remainder, and division operations are evaluated left to right.

+, − Then addition and subtraction are evaluated left to right.

==,<>,!=,<,<=,>,>= Then conditional operators are evaluated.

not The boolean operators are evaluated last.

and

or

3.12 Looping With The while Statement

Many kinds of machines, including computers, derive much of their power from the principle of repeated cycling. SAGE provides a number of ways to implement repeated cycling in a program and these ways range from straight-forward to subtle. We will begin discussing looping in SAGE by starting with the straight-forward **while** statement.

The syntax specification for the **while** statement is as follows:

```
while <expression>:
    <statement>
    <statement>
    <statement>
    .
    .
    .
```

The **while** statement is similar to the **if** statement except it will repeatedly execute the statements it contains as long as the expression to the right of its header is true. As soon as the expression returns a False object, the **while** statement skips the statements it contains and execution continues with the statement that immediately follows the **while** statement (if there is one).

The following example program uses a **while** loop to print the integers from 1 to 10:

```
# Print the integers from 1 to 10.

x = 1  #Initialize a counting variable to 1 outside of the loop.

while x <= 10:
    print x
    x = x + 1  #Increment x by 1.

    1
    2
    3
    4
    5
    6
    7
    8
    9
    10
```

In this program, a single variable called **x** is created. It is used to tell the **print** statement which integer to print and it is also used in the expression that determines if the **while** loop should continue to loop or not.

When the program is executed, 1 is placed into x and then the **while** statement is entered. The expression x <= 10 becomes 1 <= 10 and, since 1 is less than or equal to 10, a boolean object containing **True** is returned by the expression.

The **while** statement sees that the expression returned a true object and therefore it executes all of the statements inside of itself from top to bottom.

The print statement prints the current contents of x (which is 1) then x = x + 1 is executed.

The expression x = x + 1 is a standard expression form that is used in many programming languages. Each time an expression in this form is evaluated, it increases the variable it contains by 1. Another way to describe the effect this expression has on x is to say that it **increments** x by 1.

In this case x contains 1 and, after the expression is evaluated, x contains 2.

After the last statement inside of a **while** statement is executed, the **while** statement reevaluates the expression to the right of its header to determine whether it should continue looping or not. Since x is 2 at this point, the expression returns **True** and the code inside the **while** statement is executed again. This loop will be repeated until x is incremented to 11 and the expression returns **False**.

The previous program can be adjusted in a number of ways to achieve different results. For example, the following program prints the integers from 1 to 100 by increasing the 10 in the expression which is at the right side of the **while** header to 100. A comma has been placed after the print statement so that its output is displayed on the same line until it encounters the right side of the window.

```
# Print the integers from 1 to 100.

x = 1

while x <= 100:
    print x,
    x = x + 1  #Increment x by 1.

  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
  28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51
  52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
  76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99
  100
```

The following program prints the odd integers from 1 to 99 by changing the increment value in the increment expression from 1 to 2:

```
# Print the odd integers from 1 to 99.

x = 1

while x <= 100:
    print x,
    x = x + 2  #Increment x by 2.

 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51
 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
```

Finally, this program prints the numbers from 1 to 100 in reverse order:

```
# Print the integers from 1 to 100 in reverse order.

x = 100

while x >= 1:
    print x,
    x = x - 1  #Decrement x by 1.

 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 78 77
 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55 54 53
 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29
 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2
 1
```

In order to achieve this result, this program had to initialize x to 100, check to see if x was **greater than or equal to 1** (x >= 1) to continue looping, and decrement x by **subtracting** 1 from it instead of adding 1 to it.

3.13 Long-Running Loops, Infinite Loops, And Interrupting Execution

It is easy to create a loop that will execute a large number of times, or even an infinite number of times, either on purpose or by mistake. When you execute a program that contains an infinite loop, it will run until you tell SAGE to **interrupt** its execution. This is done by selecting the **Action** menu which is near the upper left part of the worksheet and then selecting the **Interrupt** menu item. Programs with long-running loops can be interrupted this way too. In both cases, the vertical green execution bar will indicate that the program is currently executing and the green bar will disappear after the program has been interrupted.

This program contains an infinite loop:

```
#Infinite loop example program.

x = 1
while x < 10:
    answer = x + 1
```

Since the contents of x is never changed inside the loop, the expression x < 10 always evaluates to True which causes the loop to continue looping.

Execute this program now and then interrupt it using the worksheet's **Interrupt** command. Sometimes simply interrupting the worksheet is not enough to stop execution and then you will need to select **Action -> Restart worksheet**. When a worksheet is **restarted**, however, **all variables are set back to their initial conditions** so the cells that assigned values to these variables will each need to be executed again.

3.14 Inserting And Deleting Worksheet Cells

If you need to insert a new worksheet cell between two existing worksheet cells, move your mouse cursor between the two cells just above the bottom one and a **horizontal blue bar** will appear. Click on this blue bar and a new cell will be inserted into the worksheet at that point.

If you want to delete a cell, delete all of the text in the cell so that it is empty. Make sure the cursor is in the now empty cell and then press the backspace key on your keyboard. The cell will then be deleted.

3.15 Introduction To More Advanced Object Types

Up to this point, we have only used objects of type 'sage.rings.integer.Integer' and of type 'str'. However, SAGE includes a large number of mathematical and nonmathematical object types that can be used for a wide variety of purposes. The following sections introduce two additional mathematical object types and two nonmathematical object types.

3.15.1 Rational Numbers

Rational numbers are held in objects of type **sage.rings.rational.Rational**. The following example prints the type of the rational number 1/2, assigns 1/2 to variable x, prints x, and then displays the type of the object that x references:

```
print type(1/2)
x = 1/2
print x
type(x)
```

```
|
    <type 'sage.rings.rational.Rational'>
    1/2
    <type 'sage.rings.rational.Rational'>
```

The following code was entered into a separate cell in the worksheet after the previous code was executed. It shows two rational numbers being added together and the result, which is also a rational number, being assigned to the variable y:

```
y = x + 3/4
print y
type(y)
|
    5/4
    <type 'sage.rings.rational.Rational'>
```

If a rational number is added to an integer number, the result is placed into an object of type sage.rings.rational.Rational:

```
x = 1 + 1/2
print x
type(x)
|
    3/2
    <type 'sage.rings.rational.Rational'>
```

3.15.2 Real Numbers

Real numbers are held in objects of type **sage.rings.real_mpfr.RealNumber**. The following example prints the type of the real number .5, assigns .5 to variable x, prints x, and then displays the type of the object that x references:

```
print type(.5)
x = .5
print x
type(x)
|
    <type 'sage.rings.real_mpfr.RealNumber'>
    0.500000000000000
    <type 'sage.rings.real_mpfr.RealNumber'>
```

The following code was entered in a separate cell in the worksheet after the previous code was executed. It shows two real numbers being added together and the result, which is also a real number, being assigned to the variable y:

```
y = x + .75
```

```
print y
type(y)
|
    1.25000000000000
    <type 'sage.rings.real_mpfr.RealNumber'>
```

If a real number is added to a rational number, the result is placed into an object of type sage.rings.real_mpfr.RealNumber:

```
x = 1/2 + .75
print x
type(x)
|
    1.25000000000000
    <type 'sage.rings.real_mpfr.RealNumber'>
```

3.15.3 Objects That Hold Sequences Of Other Objects: Lists And Tuples

The **list** object type is designed to hold other objects in an ordered collection or **sequence**. Lists are very flexible and they are one of the most heavily used object types in SAGE. Lists can hold objects of any type, they can grow and shrink as needed, and they can be nested. Objects in a list can be accessed by their position in the list and they can also be replaced by other objects. A list's ability to grow, shrink, and have its contents changed makes it a **mutable** object type.

One way to create a list is by placing 0 or more objects or expressions inside of a pair of square braces. The following program begins by printing the type of a list. It then creates a list that contains the numbers 50, 51, 52, and 53, assigns it to the variable x, and prints x.

Next, it prints the objects that are in positions 0 and 3, replaces the 53 at position 3 with 100, prints x again, and finally prints the type of the object that x refers to:

```
print type([])
x = [50,51,52,53]
print x
print x[0]
print x[3]
x[3] = 100
print x
type(x)
|
     <type 'list'>
```

```
    [50, 51, 52, 53]
    50
    53
    [50, 51, 52, 100]
    <type 'list'>
```

Notice that the first object in a list is placed at position 0 instead of position 1 and that this makes the position of the last object in the list 1 less than the length of the list. Also notice that an object in a list is accessed by placing a pair of square brackets, which contain its position number, to the right of a variable that references the list.

The next example shows that different types of objects can be placed into a list:

```
x = [1, 1/2, .75, 'Hello', [50,51,52,53]]
print x
|
   [1, 1/2, 0.750000000000000, 'Hello', [50, 51, 52, 53]]
```

Tuples are also **sequences** and are similar to lists except they are immutable. They are created using a pair of **parentheses** instead of a pair of square brackets and being **immutable** means that once a tuple object has been created, it cannot grow, shrink, or change the objects it contains.

The following program is similar to the first example list program, except it uses a tuple instead of a list, it does not try to change the object in position 4, and it uses the semicolon technique to display multiple results instead of print statements:

```
print type(())
x = (50,51,52,53)
x;x[0];x[3];x;type(x)
|
   <type 'tuple'>
   (50, 51, 52, 53)
   50
   53
   (50, 51, 52, 53)
   <type 'tuple'>
```

3.15.3.1 Tuple Packing And Unpacking

When multiple values separated by commas are assigned to a single variable, the values are automatically placed into a tuple and this is called **tuple packing**:

```
t = 1,2
```

```
t

    (1, 2)
```

When a tuple is assigned to multiple variables which are separated by commas, this is called **tuple unpacking**:

```
a,b,c = (1,2,3)
a;b;c

    1
    2
    3
```

A requirement with tuple unpacking is that the number of objects in the tuple must match the number of variables on the left side of the equals sign.

3.16 Using while Loops With Lists And Tuples

Statements that loop can be used to select each object in a list or a tuple in turn so that an operation can be performed on these objects. The following program uses a **while** loop to print each of the objects in a list:

```
#Print each object in the list.
x = [50,51,52,53,54,55,56,57,58,59]
y = 0
while y <= 9:
    print x[y]
    y = y + 1

    50
    51
    52
    53
    54
    55
    56
    57
    58
    59
```

A loop can also be used to search through a list. The following program uses a **while** loop and an **if** statement to search through a list to see if it contains the number 53. If 53 is found in the list, a message is printed.

```
#Determine if 53 is in the list.
x = [50,51,52,53,54,55,56,57,58,59]
y = 0
while y <= 9:
    if x[y] == 53:
        print "53 was found in the list at position", y
    y = y + 1
|
   53 was found in the list at position 3
```

3.17 The in Operator

Looping is such a useful capability that SAGE even has an operator called **in** that loops internally. The **in** operator is able to automatically search a list to determine if it contains a given object. If it finds the object, it will return **True** and if it doesn't find the object, it will return **False**. The following programs shows both cases:

```
print 53 in [50,51,52,53,54,55,56,57,58,59]
print 75 in [50,51,52,53,54,55,56,57,58,59]
|
   True
   False
```

The **not** operator can also be used with the **in** operator to change its result:

```
print 53 not in [50,51,52,53,54,55,56,57,58,59]
print 75 not in [50,51,52,53,54,55,56,57,58,59]
|
   False
   True
```

3.18 Looping With The for Statement

The **for** statement uses a loop to index through a list or tuple like the **while** statement does, but it is more flexible and automatic. Here is a simplified syntax specification for the **for** statement:

for <target> in <object>:
    <statement>
    <statement>
    <statement>
    .
    .
    .

In this syntax, <target> is usually a variable and <object> is usually an object that contains other objects. In the remainder of this section, lets assume that <object> is a list. The **for** statement will select each object in the list in turn, assign it to <target>, and then execute the statements that are inside its indented code block. The following program shows a **for** statement being used to print all of the items in a list:

```
for x in [50,51,52,53,54,55,56,57,58,59]:
    print x

   50
   51
   52
   53
   54
   55
   56
   57
   58
   59
```

3.19 Functions

Programming **functions** are statements that consist of named blocks of code that can be executed one or more times by being **called** from other parts of the program. Functions can have objects passed to them from the calling code and they can also return objects back to the calling code. An example of a function is the type() command which we have been using to determine the types of objects.

Functions are one way that SAGE enables code to be reused. Most programming languages allow code to be reused in this way, although in other languages these type of code reuse statements are sometimes called **subroutines** or **procedures**.

Function names use all lower case letters. If a function name contains more than one word (like calculatesum) an underscore can be placed between the words to improve readability (calculate_sum).

3.20 Functions Are Defined Using the def Statement

The statement that is used to define a function is called **def** and its syntax specification is as follows:

```
def <function name>(arg1, arg2, ... argN):
   <statement>
   <statement>
```

```
    <statement>
    .
    .
    .
```

The **def** statement contains a header which includes the function's name along with the arguments that can be passed to it. A function can have 0 or more arguments and these arguments are placed within parentheses. The statements that are to be executed when the function is called are placed inside the function using an indented block of code.

The following program defines a function called **addnums** which takes two numbers as arguments, adds them together, and returns their sum back to the calling code using a **return** statement:

```
def addnums(num1, num2):
    """
    Returns the sum of num1 and num2.
    """
    answer = num1 + num2
    return answer

#Call the function and have it add 2 to 3.
a = addnums(2, 3)
print a

#Call the function and have it add 4 to 5.
b = addnums(4, 5)
print b
```

```
    5
    9
```

The first time this function is called, it is passed the numbers 2 and 3 and these numbers are assigned to the variables **num1** and **num2** respectively. Argument variables that have objects passed to them during a function call can be used within the function as needed.

Notice that when the function returns back to the caller, the object that was placed to the right of the **return** statement is made available to the calling code. It is almost as if the function itself is replaced with the object it returns. Another way to think about a returned object is that it is sent out of the left side of the function name in the calling code, through the equals sign, and is assigned to the variable. In the first function call, the object that the function returns is being assigned to the variable 'a' and then this object is printed.

The second function call is similar to the first call, except it passes different numbers (4, 5) to the function.

3.21 A Subset Of Functions Included In SAGE

SAGE includes a large number of pre-written functions that can be used for a wide variety of purposes. Table 3 contains a subset of these functions and a longer list of functions can be found in SAGE's documentation. A more complete list of functions can be found in the *SAGE Reference Manual*.

Function Name	Description
abs	Return the absolute value of the argument.
acos	The arccosine function.
add	Returns the sum of a sequence of numbers (NOT strings) plus the value of parameter 'start'. When the sequence is empty, returns start.
additive_order	Return the additive order of x.
asin	The arcsine function.
atan	The arctangent function.
binomial	Return the binomial coefficient.
ceil	The ceiling function.
combinations	A combination of a multiset (a list of objects which may contain the same object several times) mset is an unordered selection without repetitions and is represented by a sorted sublist of mset. Returns the set of all combinations of the multiset mset with k elements.
complex	Create a complex number from a real part and an optional imaginary part. This is equivalent to (real + imag*1j) where imag defaults to 0.
cos	The cosine function.
cosh	The hyperbolic cosine function.
coth	The hyperbolic cotangent function.
csch	The hyperbolic cosecant function.
denominator	Return the denominator of x.
derivative	The derivative of f.
det	Return the determinant of x.
diff	The derivative of f.
dir	Return an alphabetized list of names comprising (some of) the attributes of the given object, and of attributes reachable from it.
divisors	Returns a list of all positive integer divisors.
dumps	Dump obj to a string s. To recover obj, use loads(s).
e	The base of the natural logarithm.
eratosthenes	Return a list of the primes <= n.
exists	If S contains an element x such that P(x) is True, this function returns True and the element x. Otherwise it returns False and None.
exp	The exponential function, exp(x) = e^x.
expand	Returns the expanded form of a polynomial.
factor	Returns the factorization of the integer n as a sorted list of tuples (p,e).
factorial	Compute the factorial of n, which is the product of 1 * 2 * 3 ... (n-1) n.

fibonacci	Returns then n-th Fibonacci number.
fibonacci_sequence	Returns an iterator over the Fibonacci sequence, for all fibonacci numbers f_n from n = start up to (but not including) n = stop.
fibonacci_xrange	Returns an iterator over all of the Fibonacci numbers in the given range, including f_n = start up to, but not including, f_n = stop.
find_root	Numerically find a root of f on the closed interval [a,b (or [b,a]) if possible, where f is a function in the one variable.
floor	The floor function.
forall	If P(x) is true every x in S, return True and None. If there is some element x in S such that P is not True, return False and x.
forget	Forget the given assumption, or call with no arguments to forget all assumptions. Here an assumption is some sort of symbolic constraint.
function	Create a formal symbolic function with the name *s*.
gaussian_binomial	Return the gaussian binomial.
gcd	The greatest common divisor of a and b.
generic_power	The m-th power of a, where m is a non-negative.
get_memory_usage	Return memory usage.
hex	Return the hexadecimal representation of an integer or long integer.
imag	Return the imaginary part of x.
imaginary	Return the imaginary part of a complex number.
integer_ceil	Return the ceiling of x.
integer_floor	Return the largest integer <= x.
integral	Return an indefinite integral of an object x.
integrate	The integral of f.
interval	Integers between a and b inclusive (a and b integers).
is_AlgebraElement	Return True if x is of type AlgebraElement.
is_commutative	
is_ComplexNumber	
is_even	Return whether or not an integer x is even, e.g., divisible by 2.
is_Functor	
is_Infinite	
is_Integer	
is_odd	Return whether or not x is odd. This is by definition the complement of is_even.
is_power_of_two	This function returns True if and only if n is a power of 2
is_prime	Returns True if x is prime, and False otherwise.
is_prime_power	Returns True if x is a prime power, and False otherwise.

is_pseudoprime	Returns True if x is a pseudo-prime, and False otherwise.
is_RealNumber	Return True if x is of type RealNumber, meaning that it is an element of the MPFR real field with some precision.
is_Set	Returns true if x is a SAGE Set.
is_square	Returns whether or not n is square, and if n is a square also returns the square root. If n is not square, also returns None.
is_SymbolicExpression	
isqrt	Return an integer square root, i.e., the floor of a square root.
laplace	Attempts to compute and return the Laplace transform of self.
latex	Use latex(...) to typeset a SAGE object.
lcm	The least common multiple of a and b, or if a is a list and b is omitted the least common multiple of all elements of v.
len	Returns the number of items of a sequence or mapping.
lim	Return the limit as the variable v approaches a from the given direction.
limit	Return the limit as the variable v approaches a from the given direction.
list	list() -> new list, list(sequence) -> new list initialized from sequence's items
list_plot	list_plot takes a single list of data, in which case it forms a list of tuples (i,di) where i goes from 0 to len(data)-1 and di is the ith data value, and puts points at those tuple values. list_plot also takes a list of tuples (dxi, dyi) where dxi is the ith data representing the x-value, and dyi is the ith y-value if plotjoined=True, then a line spanning all the data is drawn instead.
load	Load SAGE object from the file with name filename, which will have an .sobj extension added if it doesn't have one. NOTE: There is also a special SAGE command (that is not available in Python) called load that you use by typing sage: load filename.sage
loads	Recover an object x that has been dumped to a string s using s = dumps(x).
log	The natural logarithm of the real number 2.
matrix	Create a matrix.
max	With a single iterable argument, return its largest item. With two or more arguments, return the largest argument.
min	With a single iterable argument, return its smallest item. With two or more arguments, return the smallest argument.
minimal_polynomial	Return the minimal polynomial of x.
mod	
mrange	Return the multirange list with given sizes and type.
mul	Return the product of the elements in the list x.
next_prime	The next prime greater than the integer n.
next_prime_power	The next prime power greater than the integer n. If n is a prime

norm	Return the norm of x.
normalvariate	Normal distribution.
nth_prime	
number_of_arrangements	Returns the size of arrangements(mset,k).
number_of_combinations	Returns the size of combinations(mset,k).
number_of_derangements	Returns the size of derangements(mset).
number_of_divisors	Return the number of divisors of the integer n.
number_of_permutations	Returns the size of permutations(mset).
numerator	Return the numerator of x.
numerical_integral	Returns the numerical integral of the function on the interval from xmin to xmax and an error bound.
numerical_sqrt	Return a square root of x.
oct	Return the octal representation of an integer or long integer.
order	Return the order of x. If x is a ring or module element, this is the additive order of x.
parametric_plot	parametric_plot takes two functions as a list or a tuple and make a plot with the first function giving the x coordinates and the second function giving the y coordinates.
parent	Return x.parent() if defined, or type(x) if not.
permutations	A permutation is represented by a list that contains exactly the same elements as mset, but possibly in different order.
pg	Permutation groups. In SAGE a permutation is represented as either a string that defines a permutation using disjoint cycle notation, or a list of tuples, which represent disjoint cycles.
pi	The ratio of a circle's circumference to its diameter.
plot	
pow	With two arguments, equivalent to x^y. With three arguments, equivalent to (x^y) % z, but may be more efficient (e.g. for longs)
power_mod	The m-th power of a modulo the integer n.
prange	List of all primes between start and stop-1, inclusive.
previous_prime	The largest prime < n.
previous_prime_power	The largest prime power < n.
prime_divisors	The prime divisors of the integer n, sorted in increasing order.
prime_factors	The prime divisors of the integer n, sorted in increasing order.
prime_powers	List of all positive primes powers between start and stop-1, inclusive.
primes	Returns an iterator over all primes between start and stop-1, inclusive.
primes_first_n	Return the first n primes.
prod	Return the product of the elements in the list x.
quo	Return the quotient object x/y, e.g., a quotient of numbers or of a

	polynomial ring x by the ideal generated by y, etc.
quotient	Return the quotient object x/y, e.g., a quotient of numbers or of a polynomial ring x by the ideal generated by y, etc.
random	Returns a random number in the interval [0, 1].
random_prime	Returns a random prime p between 2 and n (i.e. 2 <= p <= n).
randrange	Choose a random item from range(start, stop[, step]).
range	Returns a list containing an arithmetic progression of integers.
rational_reconstruction	This function tries to compute x/y, where x/y is rational number.
real	Return the real part of x.
reduce	Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the sequence to a single value.
repr	Return the canonical string representation of the object.
reset	Delete all user defined variables, reset all globals variables back to their default state, and reset all interfaces to other computer algebra systems. If vars is specified, just restore the value of vars and leave all other variables alone (i.e., call restore).
restore	Restore predefined global variables to their default values.
round	Round a number to a given precision in decimal digits (default 0 digits). This always returns a real double field element.
sample	Chooses k unique random elements from a population sequence.
save	Save obj to the file with name filename, which will have an .sobj extension added if it doesn't have one. This will *replace* the contents of filename.
save_session	Save all variables that can be saved wto the given filename.
search	Return (True,i) where i is such that v[i] == x if there is such an i, or (False,j) otherwise, where j is the position that a should be inserted so that v remains sorted.
search_doc	Full text search of the SAGE HTML documentation for lines containing s.
search_src	Search sage source code for lines containing s.
sec	The secant function.
sech	The hyperbolic secant function.
seed	
seq	A mutable list of elements with a common guaranteed universe, which can be set immutable.
set	Build an unordered collection of unique elements.
show	Show a graphics object x.
show_default	Set the default for showing plots using the following commands: plot, parametric_plot, polar_plot, and list_plot.
shuffle	
sigma	Return the sum of the k-th powers of the divisors of n.

simplify	Simplify the expression f.
sin	The sine function.
sinh	The hyperbolic sine function.
sleep	
slice	Create a slice object. This is used for extended slicing (e.g. a[0:10:2]).
slide	Use latex(...) to typeset a SAGE object. Use %slide instead to typeset slides.
solve	Algebraically solve an equation or system of equations for given variables.
sorted	
sqrt	The square root function. This is a symbolic square root.
square_free_part	Return the square free part of x, i.e., a divisor z such that x = z y^2, for a perfect square y^2.
srange	Return list of numbers \code{a, a+step, ..., a+k*step}, where a +k*step < b and a+(k+1)*step > b. The type of the entries in the list are the type of the starting value.
str	Return a nice string representation of the object.
subfactorial	Subfactorial or rencontres numbers, or derangements: number of permutations of n elements with no fixed points.
sum	Returns the sum of a sequence of numbers (NOT strings) plus the value of parameter 'start'
super	Typically used to call a cooperative superclass method.
symbolic_expression	
sys	This module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.
tan	The tangent function.
tanh	The hyperbolic tangent function.
taylor	Expands self in a truncated Taylor or Laurent series in the variable v around the point a, containing terms through (x - a)^n.
transpose	
trial_division	Return the smallest prime divisor <= bound of the positive integer n, or n if there is no such prime.
two_squares	Write the integer n as a sum of two integer squares if possible; otherwise raise a ValueError.
type	Returns an object's type.
union	Return the union of x and y, as a list.
uniq	Return the sublist of all elements in the list x that is sorted and is such that the entries in the sublist are unique.
valuation	The exact power of p>0 that divides the integer m.
var	Create a symbolic variable with the name *s*.

vars	Without arguments, equivalent to locals(). With an argument, equivalent to object.__dict__.
vector	Return a vector over R with given entries.
version	Return the version of SAGE.
view	Compute a latex representation of each object in objects. NOTE: In notebook mode this function simply embeds a png image in the output
walltime	Return the wall time.
xgcd	Returns triple of integers (g,s,t) such that g = s*a+t*b = gcd(a,b).
xinterval	Iterator over the integers between a and b, inclusive.
xrange	Like range(), but instead of returning a list, returns an object that generates the numbers in the range on demand.
zip	Return a list of tuples, where each tuple contains the i-th element from each of the argument sequences.

Table 3: Subset of SAGE functions

3.22 Obtaining Information On SAGE Functions

Table 3 includes a list of functions along with a short description of what each one does. This is not enough information, however, to show how to actually use these functions. One way to obtain additional information on any function is to type its name followed by a question mark '?' into a worksheet cell then press the <tab> key:

```
is_even?<tab>

File: /opt/sage-2.7.1-debian-32bit-i686-
Linux/local/lib/python2.5/site-packages/sage/misc/functional.py
Type:           <type 'function'>
Definition:     is_even(x)
Docstring:

    Return whether or not an integer x is even, e.g., divisible by 2.

    EXAMPLES:
        sage: is_even(-1)
        False
        sage: is_even(4)
        True
        sage: is_even(-2)
        True
```

A gray window will then be shown which contains the following information about the function:

File: Gives the name of the file that contains the source code that implements the function. This is useful if you would like to locate the file to see how the function is implemented or to edit it.

Type: Indicates the type of the object that the name passed to the information service refers to.

Definition: Shows how the function is called.

Docstring: Displays the documentation string that has been placed into the source code of this function.

You may obtain help on any of the functions listed in Table 3, or the SAGE reference manual, using this technique. Also, if you place two question marks '??' after a function name and press the <tab> key, the function's source code will be displayed.

3.23 Information Is Also Available On User-Entered Functions

The information service can also be used to obtain information on user-entered functions and a better understanding of how the information service works can be gained by trying this at least once.

If you have not already done so in your current worksheet, type in the addnums function again and execute it:

```
def addnums(num1, num2):
    """
    Returns the sum of num1 and num2.
    """
    answer = num1 + num2
    return answer

#Call the function and have it add 2 to 3.
a = addnums(2, 3)
print a

  5
```

Then obtain information on this newly-entered function using the technique from the previous section:

```
addnums?<tab>
```

```
File: /home/sage/sage_notebook/worksheets/root/9/code/8.py
Type: <type 'function'>
Definition: addnums(num1, num2)
Docstring:

    Returns the sum of num1 and num2.
```

This shows that the information that is displayed about a function is obtained from the function's source code.

3.24 Examples Which Use Functions Included With SAGE

The following short programs show how some of the functions listed in Table 3 are used:

```
#Determine the sum of the numbers 1 through 10.
add([1,2,3,4,5,6,7,8,9,10])
|
   55

#Cosine of 1 radian.
cos(1.0)
|
   0.540302305868140

#Determine the denominator of 15/64.
denominator(15/64)
|
   64

#Obtain a list that contains all positive
#integer divisors of 20.
divisors(20)
|
   [1, 2, 4, 5, 10, 20]

#Determine the greatest common divisor of 40 and 132.
gcd(40,132)
|
   4

#Determine the product of 2, 3, and 4.
mul([2,3,4])
|
   24

#Determine the length of a list.
```

```
a = [1,2,3,4,5,6,7]
len(a)
|
   7

#Create a list which contains the integers 0 through 10.
a = srange(11)
a
|
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

#Create a list which contains real numbers between
#0.0 and 10.5 in steps of .5.
a = srange(11,step=.5)
a
|
   [0.0000000, 0.5000000, 1.000000, 1.500000, 2.000000, 2.500000,
   3.000000, 3.500000, 4.000000, 4.500000, 5.000000, 5.500000,
   6.000000, 6.500000, 7.000000, 7.500000, 8.000000, 8.500000,
   9.000000, 9.500000, 10.00000, 10.50000]

#Create a list which contains the integers -5 through 5.
a = srange(-5,6)
a
|
   [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]

#The zip() function takes multiple sequences and groups
#parallel members inside tuples in an output list.  One
#application this is useful for is creating points from
#table data so they can be plotted.
a = [1,2,3,4,5]
b = [6,7,8,9,10]
c = zip(a,b)
c
|
   [(1, 6), (2, 7), (3, 8), (4, 9), (5, 10)]
```

3.25 Using srange() And zip() With The for Statement

Instead of manually creating a sequence for use by a **for** statement, srange() can be used to create the sequence automatically:

```
for t in srange(6):
    print t,
|
```

```
    0 1 2 3 4 5
```

The **for** statement can also be used to loop through multiple sequences in parallel using the zip() function:

```
t1 = (0,1,2,3,4)
t2 = (5,6,7,8,9)
for (a,b) in zip(t1,t2):
    print a,b

    0 5
    1 6
    2 7
    3 8
    4 9
```

3.26 List Comprehensions

Up to this point we have seen that **if statements**, **for loops**, **lists**, and **functions** are each extremely powerful when used individually and together. What is even more powerful, however, is a special statement called a **list comprehension** which allows them to be used together with a minimum amount of syntax.

Here is the simplified syntax for a list comprehension:

[expression **for** *variable* **in** *sequence* [**if** *condition*] **]**

What a list comprehension does is to loop through a *sequence* placing each sequence member into the specified *variable* in turn. The expression also contains the *variable* and, as each member is placed into the *variable*, the *expression* is evaluated and the result is placed into a new list. When all of the members in the sequence have been processed, the new list is returned.

In the following example, **t** is the variable, **2*t** is the expression, and **[1,2,3,4,5]** is the sequence:

```
a = [2*t for t in [0,1,2,3,4,5]]
a

    [0, 2, 4, 6, 8, 10]
```

Instead of manually creating the sequence, the srange() function is often used to create it automatically:

```
a = [2*t for t in srange(6)]
a

    [0, 2, 4, 6, 8, 10]
```

An optional **if** statement can also be used in a list comprehension to filter the results that are placed in the new list:

```
a = [b^2 for b in range(20) if b % 2 == 0]
a

    [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

In this case, only results that are evenly divisible by 2 are placed in the output list.

4 Object Oriented Programming

The purpose of this chapter is to **introduce the main concepts** behind how object oriented SAGE code works and how it is used to solve problems. It assumes that you have little or no Object Oriented Programming (OOP) experience and it is going to give you enough of an understanding of OOP so that you can more effectively use SAGE objects to solve problems.

Do not worry too much if this OOP stuff does not completely sink in right away because you can use SAGE objects to solve problems without yet having the skill needed to program objects from scratch yourself. Having said that, this chapter does show how to program an object from scratch so you can better understand how SAGE's pre-built objects work.

4.1 Object Oriented Mind Re-wiring

In my opinion, one of the more difficult things you will do in the area of programming is to make the mental switch from the procedural programming paradigm to the object oriented programming paradigm. The problem is not that object oriented programming is necessarily more difficult than procedural programming. The problem is that it is so different in its approach to solving programming problems that some mental re-wiring is going to have to happen before you truly "get it". This mental re-wiring is a process that happens very slowly as you write object oriented programs and dig deeper into object oriented books in an effort to really understand what OOP is all about.

Right from the beginning you will see that there is something very special and powerful going on, but it will elude your efforts to firmly grasp it. When you do finally "get it" it will usually not come all at once like a bright light going on. It is more like a dim light that you can sense glowing in the back of your mind that brightens very slowly. For each new programming problem you encounter, the front part of your mind will still produce a procedural plan to solve it. However you will begin to notice that this glow in the back of your mind will present object oriented strategies (dim at first, but slowly increasing in clarity) that will also solve the problem and these object oriented strategies are so interesting that over time you will find yourself paying more and more attention to them. Eventually a time will come when many programming problems will trigger the production of rich object oriented strategies for solving them from the now bright object oriented part of your mind.

4.2 Attributes And Behaviors

Object oriented programming is a software design philosophy where software is made to work similar to the way that objects in the physical world work. All physical objects have **attributes** and **behaviors**. One example is a typical office chair which has **color**, **number of wheels**, and **material type** as attributes and

spin, **roll**, and **set height** as behaviors.

Software objects are made to work like physical objects and so they also have attributes and behaviors. A software object's **attributes** are held in special variables called **instance variables** and its **behaviors** are determined by **code** which is held in **methods** (which are also called **member functions**). **Methods** are similar to standard functions except they are associated with objects instead of "floating around free". In SAGE, instance variables and methods are often referred to as just **attributes**.

After an object is created, it is used by sending it **messages**, which means to **call** or **invoke** its methods. In the case of the chair, we could imagine sending it a **chair.spin(3)** message which would tell the chair to spin 3 times, or a **chair.setHeight(32)** message which would tell the chair to set its height to 32 centimeters.

4.3 Classes (Blueprints That Are Used To Create Objects)

A **class** can be thought of as a **blueprint** that is used to construct objects and it is conceptually similar to a house blueprint. An architect uses a blueprint to precisely define exactly how a given house should be constructed, what materials should be used, what its various dimensions should be, etc. After the blueprint is finished, it can be used to construct one house or many houses because the blueprint contains the information that describes how to create a house, it is not the house itself. A programmer creating a **class** is very similar to an architect creating a **house blueprint** except that the architect uses a drafting table or a CAD system to develop a blueprint while a programmer uses a text editor or an IDE (Integrated Development Environment) to develop a class.

4.4 Object Oriented Programs Create And Destroy Objects As Needed

The following analogy describes how software objects are created and destroyed as needed in object oriented program. Creating an object is also called **instantiating** it because the **class** (blueprint) that defines the object is being used to create an object instance. The act of destroying an object and reclaiming the memory and other resources it was using is called **garbage collection**.

Imagine that a given passenger jet can operate in a manner which is similar an object oriented program and that the jet is being prepared to fly across the Atlantic ocean from New York to London. Just before takeoff, the blueprints for every part of the aircraft are brought to the tarmac and given to a team of workers who will use them to very quickly construct all of the components needed to build the aircraft. As each component is constructed, it is attached to the proper place on the aircraft and in a short time the aircraft is complete and ready to use. The passengers are loaded onto the jet and and it takes off.

After the plane leaves the ground, the landing gear are disintegrated (garbage collected) because they are not needed during the flight and hauling them across the Atlantic ocean would just waste costly fuel. There is no need to worry, however, because the landing gear will be reconstructed using the proper blueprints (classes) just before landing in London

A few minutes after takeoff the pilot receives notification that the company that manufactured the aircraft's jet engines has just released a new model that is 15% more fuel efficient than the ones that the aircraft is currently using and the airline is going to upgrade the aircraft's engines while the plane is in flight. The airline sends the blueprints for the new engines over the network to the plane and these are used to construct (instantiate) three of the new engines. After the new engines are constructed, the three old engines are shut down one at a time, replaced with a new engine, and disintegrated. The engine upgrade goes smoothly and the passengers are not even aware that the upgrade took place.

This flight just happens to have an important world figure on board and halfway through the flight a hostile aircraft is encountered which orders our pilot to change his course. Instead of complying with this demand, however, the pilot retrieves a set of blueprints from the blueprint library for a 50mm machine gun turret, has 4 of these turrets constructed , and then has them attached to the plane's top, bottom, nose, and tail sections. A few blasts from one of these guns is enough to deter the hostile aircraft and it quickly moves away, eventually dropping off of the radar screen. The rest of the flight is uneventful. As the aircraft approaches London, the machine gun turrets are disintegrated, a new set of landing gear are constructed using the landing gear blueprints, and the plane safely lands. After the passengers are in the terminal, the whole plane is disintegrated.

4.5 Object Oriented Program Example

The following two sections cover a simple object oriented program called **Hellos**. The first section presents a version of the program which does not contain any comments so the code itself is easier to see. The second section contains a fully-commented version of the program along with a detailed description of how the program works.

4.5.1 Hellos Object Oriented Program Example (No Comments)

```
class Hellos:

  def __init__(self, mess):
      self.message = mess
```

```
  def print_message(self):
      print"The message is: ", self.message

  def say_goodbye(self):
      print "Goodbye!"

  def print_hellos(self, total):
      count = 1
      while count <= total:
          print"Hello ", count
          count = count + 1

      print " "


obj1 = Hellos("Are you having fun yet?")
obj2 = Hellos("Yes I am!")

obj1.print_message()
obj2.print_message()
print " "

obj1.print_hellos(3)
obj2.print_hellos(5)

obj1.say_goodbye()
obj2.say_goodbye()

   The message is:  Are you having fun yet?
   The message is:  Yes I am!

   Hello  1
   Hello  2
   Hello  3

   Hello  1
   Hello  2
   Hello  3
   Hello  4
   Hello  5

   Goodbye!
   Goodbye!
```

4.5.2 Hellos Object Oriented Program Example (With

Comments)

We will now look at the **Hellos** program in more detail. This version of the program has had comments added to it. The line numbers and colons on the left side of the program are not part of the program itself and they have been added to make referencing different parts of the program easier.

```
 1:class Hellos:
 2:    """
 3:    Hellos is a 'class' and a class is a blueprint for creating
 4:    objects.  Classes consist of instance variables (attributes)
 5:    and methods (behaviors).
 6:    """
 7:
 8:    def __init__(self, mess):
 9:        """
10:        __init__ is a special kind of built-in method called a
11:        constructor.  A constructor method is only invoked once
12:        when an object is being created and its job is to complete
13:        the construction of the object.  After the object has
14:        been created its constructors are no longer used.  The
15:        purpose of this constructor is to create an instance
16:        variable called 'message' and then initialize it with a
17:        string.
18:        """
19:
20:        """
21:        This code creates an instance variable.  Every object
22:        instance created from this class 'blueprint' will have
23:        its own unique copy of any instance variables.  Instance
24:        variables hold an object's attributes (or state).
25:        The self variable here holds a reference to the current
26:        object.
27:        """
28:        self.message = mess;
29:
30:
31:
32:    def print_message(self):
33:        """
34:        print_message is an instance method that gives objects
35:        created using this class their 'print message' behavior.
36:        """
37:        print"The message is: ", self.message
38:
39:
40:
```

```
    def say_goodbye(self):
        """
        say_goodbye is an instance method that gives objects
        created using this class their 'say goodby' behavior.
        """
        print "Goodbye!"



    def print_hellos(self, total):
        """
        print_hellos is an instance method that takes the number
        of Hellos to print as an argument and it prints this many
        Hellos to the screen.
        """
        count = 1
        while count <= total:
            print"Hello ", count
            count = count + 1

        print " "


"""
The following code creates two separate Hellos objects (instances)
which are referenced by the variables obj1 and obj2 respectively.
A unique String parameter is passed to each object when it is
instantiated and this String is used to initialize the object's
state.

After the objects are created, messages are sent to them by
calling their methods in order to have them perform behaviors.
This is done by 'picking an object up' by its reference (lets
say obj1) placing a dot after this reference and then typing the
name of an object's method that you want to invoke.
"""

obj1 = Hellos("Are you having fun yet?")
obj2 = Hellos("Yes I am!")

obj1.print_message()
obj2.print_message()
print " "

obj1.print_hellos(3)
obj2.print_hellos(5)

```

```
88:obj1.say_goodbye()
89:obj2.say_goodbye()
```

On line 1 the class Hellos is defined using a **class** statement and by convention class names start with a capital letter. If the class name consists of multiple words, then the first letter of each word is capitalized and all other letters are typed in lower case (for example, HelloWorld). The class begins on line 1 and ends on line 61, which is the last line of indented code it contains. All **methods** and **instance variables** that are part of a class need to be inside the class's indented code block.

The Hellos class contains one **constructor** method on line 8, one **instance variable** which is created on line 28, and three **instance methods** on lines 32, 41, and 50 respectively. The purpose of **instance variables** are to give an object unique **attributes** that differentiate it from other objects that are created from a given class The purpose of **instance methods** are to give each object its **behaviors**. All methods in an object have access to that object's instance variables and these instance variables can be accessed by the code in these methods. Instance variable names follow the same convention that function names do.

The method on line 8 is a special method called a **constructor**. A **constructor** method is only invoked when an object is being created and its purpose is to complete the construction of the object. After the object has been created, its constructor is no longer used. The purpose of the constructor on line 8 is to initialize each Hellos object's **message** instance variable with a string that is passed to it when a new object of type Hellos is created (see lines 78 and 79).

All instance methods have an argument passed to them which contains a reference to the specific object that the method was called from. This argument is always placed into the leftmost argument position and, by convention, the variable that is placed in this position is called **self**. The **self** variable is then used to create and access that specific object's instance variables.

On line 28, the code **self.message = mess** takes the object that was passed into the constructor's **mess** variable and assigns it to an instance variable called **message**. An instance variable is created via assignment just like normal variables are. The **dot operator** '.' is used to access an object's instance variables by placing it between a variable which holds a reference to the object and the instance variable's name (like self.message or obj1.message).

The methods on lines 32, 41, and 50 give objects created using the Hellos class their behaviors. The print_message() method provides the behavior of printing the string that is present in the object's **message** instance variable and the say_goodbye() method provides the behavior of printing the string "Goodbye!" The print_hellos() method takes an integer number as a parameter and it prints the word 'Hello' that many times. The naming convention for methods is the same as the one used for function names.

The code below the Hellos class creates two separate objects (instances) which are then assigned to the variables **obj1** and **obj2** respectively. An object is created by typing its class name followed by a pair of parentheses. Any arguments that are placed within the parentheses will be passed to the constructor method.

When the Hellos class is called, a string is passed to its constructor method and this string is used to initialize the object's **state**. An object's state is determined by the contents of its instance variables. If any of an object's instance variables are changed, then the object's state has been changed too. Since Hellos objects only have one instance variable called **message**, their state is determined by this variable.

After objects are created, their behaviors are requested by calling their methods. This is done by "picking an object up" by a variable that references it (lets say obj1), placing a dot after this variable, and then typing the name of one of the object's methods that you want to invoke, followed by its arguments in parentheses.

4.6 SAGE Classes And Objects

While SAGE's functions contain many capabilities, most of SAGE's capabilities are contained in **classes** and the **objects** that are instantiated from these classes. SAGE's classes and objects represent a significant amount of information which will take a while to explain. However, the easier material will be presented first so that you can start working with SAGE objects as soon as possible.

4.7 Obtaining Information On SAGE Objects

Type the following code into a cell and execute it:

```
x = 5
print type(x)

    <type 'sage.rings.integer.Integer'>
```

We have already used the type() function to determine the type of an integer, but now we can explain what a type is in more detail. Enter sage.rings.integer.Integer followed by a question mark '?' into a new cell and then press the <tab> key:

sage.rings.integer.Integer?<tab>

```
File:/opt/sage-2.7.1-debian-32bit-i686-
Linux/local/lib/python2.5/site-packages/sage/rings/integer.so
Type:   <type 'sage.rings.integer.Integer'>
Definition: sage.rings.integer.Integer([noargspec])

Docstring:

        The class{Integer} class represents arbitrary precision
        integers.  It derives from the class{Element} class, so
        integers can be used as ring elements anywhere in SAGE.

        begin{notice}
        The class class{Integer} is implemented in Pyrex,
        as a wrapper of the GMP mpz_t integer type.
        end{notice}
```

This information indicates that sage.rings.integer.Integer is really a class that is able to create Integer objects. Also, if you place two questions marks '??' after a class name and press the <tab> key, the class's source code will be displayed.

Now, in a separate cell type x. and then press the <tab> key:

```
x.<tab>
```

```
x.additive_order          x.gcd                    x.numerator
x.base_base_extend        x.inverse_mod            x.ord
x.inverse_of_unit         x.order                  x.parent
x.base_extend             x.is_nilpotent           x.plot
x.base_extend_canonical   x.is_one                 x.powermodm_ui
x.is_perfect_power        x.powermod               x.quo_rem
x.base_extend_recursive   x.is_power               x.rename
x.base_ring               x.is_power_of            x.reset_name
x.binary                  x.is_prime               x.save
x.category                x.is_prime_power         x.set_si
x.ceil                    x.is_pseudoprime         x.set_str
x.coprime_integers        x.is_square              x.sqrt
x.crt                     x.is_squarefree          x.sqrt_approx
x.db                      x.is_unit                x.square_free_part
x.degree                  x.is_zero                x.str
x.denominator             x.isqrt                  x.substitute
x.digits                  x.jacobi                 x.test_bit
x.div                     x.kronecker              x.val_unit
x.lcm                     x.subs                   x.valuation
x.divides                 x.leading_coefficient    x.version
x.dump                    x.list                   x.xgcd
x.dumps                   x.mod                    x.parent
```

```
x.exact_log                  x.multiplicative_order  x.plot
x.factor                     x.next_prime            x.rename
x.factorial                  x.next_probable_prime   x.reset_name
x.floor                      x.nth_root              x.powermodm_ui
```

A gray window will be displayed which contains all of the methods that the object contains. If any of these methods is selected with the mouse, its name will be placed into the cell after the dot operator as a convenience. For now, select the **is_prime** method. When its name is placed into the cell, type a question mark '?' after it and press the <tab> key in order to obtain information on this method:

```
x.is_prime?
|
File:       /opt/sage-2.7.1-debian-32bit-i686-Linux/local/lib/python/
site-packages/sage/rings/integer/pyx
Type:       <type 'builtin_function_or_method '>
Definition: x.is_prime()

Docstring:

            Retuns True if self is prime

            EXAMPLES:
                sage: z = 2^31 - 1
                sage: z.is_prime()
                True
                sage: z = 2^31
                sage: z.is_prime()
                False
```

The Definition section indicates that the is_prime() method is called without passing any arguments to it and the Docstring section indicates that the method will return True if the object is prime. The following code shows the variable x (which still contains 5) being used to call the is_prime() method:

```
x.is_prime()
|
   True
```

4.8 The List Object's Methods

Lists are objects and therefore they contain methods that provide useful capabilities:

```
a = []
a.<tab>
```

```
a.append   a.extend   a.insert   a.remove   a.sort
a.count    a.index    a.pop      a.reverse
```

The following programs demonstrate some of a list object's methods:

```
# Append an object to the end of a list.
a = [1,2,3,4,5,6]
print a
a.append(7)
print a
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6, 7]
```

```
# Insert an object into a list.
a = [1,2,4,5]
print a
a.insert(2,3)
print a
```

```
[1, 2, 4, 5]
[1, 2, 3, 4, 5]
```

```
# Sort the contents of a list.
a = [8,2,7,1,6,4]
print a
a.sort()
print a
```

```
[8, 2, 7, 1, 6, 4]
[1, 2, 4, 6, 7, 8]
```

4.9 Extending Classes With Inheritence

Object technologies are subtle and powerful. They possess a number of mechanisms for dealing with complexity and **class inheritance** is one of them. **Class inheritance** is the ability of a class to obtain or **inherit** all of the instance variables and methods of another class (called a **parent class**, **super class**, or **base class**) using a minimal amount of code. A class that inherits from a parent class is called a **child class** or **sub class**. This means that a child class can do everything its parent can do along with any additional functionality that is programmed into the child.

The following program demonstrates class inheritance by having a **Person** class inherit from the built-in **object** class and having an **ArmyPrivate** class inherit from the Person class:

```
class Person(object):
    def __init__(self):
        self.rank = "I am just a Person, I have no rank."

    def __str__(self):
        return "str: " + self.rank

    def __repr__(self):
        return "repr: " + self.rank


class ArmyPrivate(Person):
    def __init__(self):
        self.rank = "ArmyPrivate."


a = object()
print type(a)

b = Person()
print type(b)

c = ArmyPrivate()
print type(c)
|
   <type 'object'>
   <class '__main__.Person'>
   <class '__main__.ArmyPrivate'>
```

After the classes have been created, this program instantiates an object of type **object** which is assigned to variable 'a', an object of type **Person** which is assigned to variable 'b', and an object of type **ArmyPrivate** which is assigned to variable 'c'.

The following code can be used to display the inheritance hierarchy of any object. If it is executed in a separate cell after the above program has been executed, the inheritance hierarchy of the ArmyPrivate class is displayed (**don't worry about trying to understand how this code works. Just use it for now.**):

```
#Display the inheritance hierarchy of an object.  Note: don't worry
#about trying to understand how this program works.  Just use it for
#now.
def class_hierarchy(cls, indent):
```

```
    print '.'*indent, cls
    for supercls in cls.__bases__:
        class_hierarchy(supercls, indent+1)

def instance_hierarchy(inst):
    print 'Inheritance hierarchy of', inst
    class_hierarchy(inst.__class__, 3)

z = ArmyPrivate()

instance_hierarchy(z)

   Inheritance hierarchy of str: ArmyPrivate
   ... <class '__main__.ArmyPrivate'>
   .... <class '__main__.Person'>
   ..... <type 'object'>
```

The instance_hierarchy function will display the inheritance hierarchy of any object that is passed to it. In this case, an ArmyPrivate object was instantiated and passed to the instance_hierarchy function and the object's inheritance hierarchy was displayed. Notice that the topmost class in the hierarchy, which is the **object** class, was printed last and that **Person** inherits from **object** and **ArmyPrivate** inherits from **Person**.

4.10 The object Class, The dir() Function, And Built-in Methods

The **object** class is built into SAGE and it contains a small number of useful methods. These methods are so useful that many SAGE classes inherit from the **object** class either 1) directly or 2) indirectly by inheriting from a class that inherits from the object class. Lets begin our discussion of the inheritance program by looking at the methods that are included in the **object** class. The **dir()** function lists all of an object's attributes (which means both its instance variables and its methods) and we can use it to see which methods an object of type **object** contains:

```
dir(a)

   ['__class__', '__delattr__', '__doc__',
   '__getattribute__','__hash__','__init__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__','__setattr__', '__str__']
```

Names which begin and end with double underscores '__' are part of SAGE and the underscores make it unlikely that these names will conflict with programmer defined names. The Person class inherits all of these attributes from the **object** class, but it only uses some of them. When a method is inherited from a parent class, the child class can either use the parent's implementation of that method or it can redefine it so that it behaves differently than the parent's version.

As discussed earlier, the __init__ method is a constructor and it helps to complete construction of each new object that is created using the class it is in. The Person class redefines the __init__ method so that it creates an instance variable called **rank** and assigns the string "I am just a Person, I have no rank" to it.

The **__ repr__** and **__str__** methods are also redefined in the Person class. The __repr__ method returns a string representation of the object it is a part of:

```
b
|
   repr: I am just a Person, I have no rank.
```

The __str__ function also returns a string representation of the object it is a part of, but only when it is passed to statements like print:

```
print b
|
   str: I am just a Person, I have no rank.
```

The __str__ method is usually used to provide a more user friendly string than the __repr__ method does but in this example, very similar strings are returned.

4.11 The Inheritance Hierarchy Of The sage.rings.integer.Integer Class

The following code displays the inheritance hierarchy of the sage.rings.integer.Integer class:

```
#Display the inheritance hierarchy of an object.  Note: don't worry
#about trying to understand how this program works.  Just use it for
#now.
def class_hierarchy(cls, indent):
    print '.'*indent, cls
    for supercls in cls.__bases__:
        class_hierarchy(supercls, indent+1)

def instance_hierarchy(inst):
    print 'Inheritance hierarchy of', inst
    class_hierarchy(inst.__class__, 3)

instance_hierarchy(1)
|
```

```
   Inheritance hierarchy of 1
   ... <type 'sage.rings.integer.Integer'>
   .... <type 'sage.structure.element.EuclideanDomainElement'>
   ..... <type 'sage.structure.element.PrincipalIdealDomainElement'>
   ...... <type 'sage.structure.element.DedekindDomainElement'>
   ....... <type 'sage.structure.element.IntegralDomainElement'>
   ........ <type 'sage.structure.element.CommutativeRingElement'>
   ......... <type 'sage.structure.element.RingElement'>
   .......... <type 'sage.structure.element.ModuleElement'>
   ........... <type 'sage.structure.element.Element'>
   ............ <type 'sage.structure.sage_object.SAGEObject'>
   ............. <type 'object'>
```

In the following explanation, I am going to leave out the beginning "sage.xxx.xxx..." part of the class names to save space. The output from the **instance_hierarchy** function indicates that the number 1 is an object of type **Integer**. It then shows that Integer inherits from **EuclideanDomainElement**, **EuclideanDomainElement** inherits from **PrincipalIdealDomainElement**, etc. At the top of the hierarchy (which is at the bottom of the list) SAGEObject inherits from object.

Here is the inheritance hierarchy for two other commonly used SAGE objects:

```
instancehierarchy(1/2)

   Inheritance hierarchy of 1/2
   ... <type 'sage.rings.rational.Rational'>
   .... <type 'sage.structure.element.FieldElement'>
   ..... <type 'sage.structure.element.CommutativeRingElement'>
   ...... <type 'sage.structure.element.RingElement'>
   ....... <type 'sage.structure.element.ModuleElement'>
   ........ <type 'sage.structure.element.Element'>
   ......... <type 'sage.structure.sage_object.SAGEObject'>
   .......... <type 'object'>
```

```
instancehierarchy(1.2)

   Inheritance hierarchy of 1.20000000000000
   ... <type 'sage.rings.real_mpfr.RealNumber'>
   .... <type 'sage.structure.element.RingElement'>
   ..... <type 'sage.structure.element.ModuleElement'>
   ...... <type 'sage.structure.element.Element'>
   ....... <type 'sage.structure.sage_object.SAGEObject'>
   ........ <type 'object'>
```

4.12 The "Is A" Relationship

Another aspect to the concept of inheritance is that, since a child class can do anything its parent can do, it can be used any place its parent object can be used. Take a look at the inheritance hierarchy of the Integer class. This hierarchy indicates that **Integer** **is a** **EuclideanDomainElement** and **EuclideanDomainElement** **is a** **PrincipalIdealDomainElement** and **PrincipalIdealDomainElement** **is a** **DedekindDomainElement** etc. until finally **SAGEObject** **is an** **object** (just like almost all the other classes are in SAGE since the object class is the root class from which they all descend). A more general way to look at this is to say a child class can be used any place any of its ancestor classes can be used.

4.13 Confused?

This chapter was probably confusing for you but again, don't worry about that. The rest of this book will contain examples which show how objects are used in SAGE and the more you see objects being used, the more comfortable you will become with them.

5 Miscellaneous Topics

5.1 Referencing The Result Of The Previous Operation

When working on a problem that spans multiple cells in a worksheet, it is often desirable to reference the result of the previous operation. The underscore symbol '_' is used for this purpose as shown in the following example:

```
2 + 3
|
   5
_
|
   5

_ + 6
|
   11

a = _ * 2
a
|
   22
```

5.2 Exceptions

In order to assure that SAGE programs have a uniform way to handle exceptional conditions that might occur while they are running, an exception display and handling mechanism is built into the SAGE platform. This section covers only displayed exceptions because exception handling is an advanced topic that is beyond the scope of this document.

The following code causes an exception to occur and information about the exception is then displayed:

```
1/0
|
   Exception (click to the left for traceback):
   ...
   ZeroDivisionError: Rational division by zero
```

Since 1/0 is an undefined mathematical operation, SAGE is unable to perform the calculation. It stops execution of the program and generates an exception to inform other areas of the program or the user about this problem. If no other part of the program handles the exception, a text explanation of the exception is displayed. In this case, the exception informs the user that a ZeroDivisionError has occurred and that this was caused by an attempt to perform "rational division by zero".

Most of the time, this is enough information for the user to locate the problem in the source code and fix it. Sometimes, however, the user needs more information in order to locate the problem and therefore the exception indicates that if the mouse is clicked to the left of the displayed exception text, additional information will be displayed:

```
Traceback (most recent call last):
  File "", line 1, in
  File "/home/sage/sage_notebook/worksheets/tkosan/2/code/2.py",
     line 4, in
    Integer(1)/Integer(0)
  File "/opt/sage-2.8.3-linux-32bit-debian-4.0-i686-
     Linux/data/extcode/sage/", line 1, in

  File "element.pyx", line 1471, in element.RingElement.__div__
  File "element.pyx", line 1485, in element.RingElement._div_c
  File "integer.pyx", line 735, in integer.Integer._div_c_impl
  File "integer_ring.pyx", line 185, in
integer_ring.IntegerRing_class._div
ZeroDivisionError: Rational division by zero
```

This additional information shows a trace of all the code in the SAGE library that was in use when the exception occurred along with the names of the files that hold the code. It allows an expert SAGE user to look at the source code if needed in order to determine if the exception was caused by a bug in SAGE or a bug in the code that was entered.

5.3 Obtaining Numeric Results

One sometimes needs to obtain the numeric approximate of an object and SAGE provides a number of ways to accomplish this. One way is to use the **n() function** and another way is to use the **n() method**. The following example shows both of these being used:

```
a = 3/4
print a
print n(a)
print a.n()

    3/4
    0.750000000000000
    0.750000000000000
```

The number of digits returned can be adjusted by using the **digits** parameter:

```
a = 3/4
print a.n(digits=30)

    0.750000000000000000000000000000
```

and the number of bits of precision can be adjusted by using the **prec** parameter:

```
a = 4/3
print a.n(prec=2)
print a.n(prec=3)
print a.n(prec=4)
print a.n(prec=10)
print a.n(prec=20)

    1.5
    1.2
    1.4
    1.3
    1.3333
```

5.4 Style Guide For Expressions

Always surround the following binary operators with a single space on either side: assignment '=', augmented assignment (+=, −=, etc.), comparisons (==, <, >, !=, <>, <=, >=, in, not in, is, is not), Booleans (and, or, not).

Use spaces around the + and − arithmetic operators and no spaces around the * , /, %, and ^ arithmetic operators:

    x = x + 1

    x = x*3 − 5%2

    c = (a + b)/(a − b)

Do not use spaces around the equals sign '=' when used to indicate a keyword argument or a default parameter value:

    a.n(digits=5)

5.5 Built-in Constants

SAGE has a number of mathematical constants built into it and the following is a list of some of the more common ones:

Pi, pi: The ratio of the circumference to the diameter of a circle.

E, e: Base of the natural logarithm.

I, i: The imaginary unit quantity.

log2: The natural logarithm of the real number 2.

Infinity, infinity: Can have + or − placed before it to indicate positive or negative infinity.

The following examples show constants being used:

```
a = pi.n()
b = e.n()
c = i.n()
a,b,c

   (3.14159265358979, 2.71828182845905, 1.00000000000000*I)
```

```
r = 4
a = 2*pi*r
a,a.n()

(8*pi, 25.1327412287183)
```

Constants in SAGE are defined as global variables and a **global variable** is a variable that is accessible by most SAGE code, including inside of functions and methods. Since constants are simply variables that have a constant object assigned to them, the variables can be reassigned if needed but then the constant object is lost. If one needs to have a constant reassigned to the variable it is normally associated with, the **restore()** function can be used. The following program shows how the variable **pi** can have the object 7 assigned to it and then have its default constant assigned to it again by passing its name inside of quotes to the **restore()** function:

```
print pi.n()

pi = 7
print pi

restore('pi')
```

```
print pi.n()
|
   3.14159265358979
   7
   3.14159265358979
```

If the restore() function is called with no parameters, all reassigned constants are restored to their original values.

5.6 Roots

The sqrt() function can be used to obtain the square root of a value, but a more general technique is used to obtain other roots of a value. For example, if one wanted to obtain the cube root of 8:

$$\sqrt[3]{8}$$

8 would be raised to the 1/3 power:

```
8^(1/3)
|
   2
```

Due to the order of operations, the rational number 1/3 needs to be placed within parentheses in order for it to be evaluated as an exponent.

5.7 Symbolic Variables

Up to this point, all of the variables we have used have been created during assignment time. For example, in the following code the variable **w** is created and then the number **8** is assigned to it:

```
w = 7
w
|
   7
```

But what if you needed to work with variables that are not assigned to any specific values? The following code attempts to print the value of the variable z, but z has not been assigned a value yet so an exception is returned:

```
print z
|
   Exception (click to the left for traceback):
   ...
   NameError: name 'z' is not defined
```

In mathematics, "unassigned variables" are used all the time. Since SAGE is mathematics oriented software, it has the ability to work with unassigned variables. In SAGE, unassigned variables are called **symbolic variables** and they are defined using the **var()** function. When a worksheet is first opened, the variable **x** is automatically defined to be a symbolic variable and it will remain so unless it is assigned another value in your code.

The following code was executed on a newly-opened worksheet:

```
print x
type(x)
|
   x
   <class 'sage.calculus.calculus.SymbolicVariable'>
```

Notice that the variable **x** has had an object of type **SymbolicVariable** automatically assigned to it by the SAGE environment.

If you would like to also use **y** and **z** as symbolic variables, the **var()** function needs to be used to do this. One can either enter **var('x,y')** or **var('x y')**. The **var()** function is designed to accept one or more variable names inside of a string and the names can either be separated by **commas** or **spaces**.

The following program shows **var()** being used to initialize **y** and **z** to be symbolic variables:

```
var('y,z')
y,z
|
   (y, z)
```

After one or more symbolic variables have been defined, the **reset()** function can be used to undefine them:

```
reset('y,z')
y,z
|
   Exception (click to the left for traceback):
   ...
   NameError: name 'y' is not defined
```

5.8 Symbolic Expressions

Expressions that contain symbolic variables are called **symbolic expressions**.

In the following example, **b** is defined to be a symbolic variable and then it is used to create the symbolic expression **2*b**:

```
var('b')
type(2*b)
|
   <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

As can be seen by this example, the symbolic expression **2*b** was placed into an object of type **SymbolicArithmetic**. The expression can also be assigned to a variable:

```
m = 2*b
type(m)
|
   <class 'sage.calculus.calculus.SymbolicArithmetic'>
```

The following program creates two symbolic expressions, assigns them to variables, and then performs operations on them:

```
m = 2*b
n = 3*b
m+n, m-n, m*n, m/n
|
   (5*b, -b, 6*b^2, 2/3)
```

Here is another example that multiplies two symbolic expressions together:

```
m = 5 + b
n = 8 + b
y = m*n
y
|
   (b + 5)*(b + 8)
```

5.9 Expanding And Factoring

If the expanded form of the expression from the previous section is needed, it is easily obtained by calling the **expand()** method (this example assumes the cells in the previous section have been run):

```
z = y.expand()
z
|
   b^2 + 13*b + 40
```

The **expanded** form of the expression has been assigned to variable **z** and the **factored** form can be obtained from **z** by using the **factor()** method:

```
z.factor()
|
   (b + 5)*(b + 8)
```

By the way, a number can be factored without being assigned to a variable by placing parentheses around it and calling its factor() method:

```
(90).factor()
|
   2 * 3^2 * 5
```

5.10 Miscellaneous Symbolic Expression Examples

```
var('a,b,c')

(5*a + b + 4*c) + (2*a + 3*b + c)
|
   5*c + 4*b + 7*a

(a + b) - (x + 2*b)
|
   -x - b + a

3*a^2 - a*(a -5)
|
   3*a^2 - (a - 5)*a

_.factor()
|
   a*(2*a + 5)
```

5.11 Passing Values To Symbolic Expressions

If values are passed to a symbolic expressions, they will be evaluated and a result will be returned. If the expression only has one variable, then the value can simply be passed to it as follows:

```
a = x^2
a(5)
|
   25
```

However, if the expression has two or more variables, each variable needs to have a value assigned to it by name:

```
var('y')
a = x^2 + y
a(x=2, y=3)
|
   7
```

5.12 Symbolic Equations and The solve() Function

In addition to working with symbolic expressions, SAGE is also able to work with **symbolic equations**:

```
var('a')
type(x^2 == 16*a^2)
|
   <class 'sage.calculus.equations.SymbolicEquation'>
```

As can be seen by this example, the symbolic equation **x^2 == 16*a^2** was placed into an object of type **SymbolicEquation**. A symbolic equation needs to use double equals '==' so that it can be assigned to a variable using a single equals '=' like this:

```
m = x^2 == 16*a^2
m, type(m)
|
  (x^2 == 16*a^2, <class 'sage.calculus.equations.SymbolicEquation'>)
```

Many symbolic equations can be solved algebraically using the **solve()** function:

```
solve(m, a)
|
   [a == -x/4, a == x/4]
```

The first parameter in the solve() function accepts a symbolic equation and the second parameter accepts the symbolic variable to be solved for.

The **solve()** function can also solve simultaneous equations:

```
var('i1,i2,i3,v0')

a = (i1 - i3)*2 + (i1 - i2)*5 + 10 - 25 == 0
b = (i2 - i3)*3 + i2*1 - 10 + (i2 - i1)*5 == 0
c = i3*14 + (i3 - i2)*3 + (i3 - i1)*2 - (-3*v0) == 0
```

```
d = v0 == (i2 - i3)*3
```

```
solve([a,b,c,d], i1,i2,i3,v0)
|
   [[i1 == 4, i2 == 3, i3 == -1, v0 == 12]]
```

Notice that, when more than one equation is passed to solve(), they need to be placed into a list.

5.13 Symbolic Mathematical Functions

SAGE has the ability to define functions using mathematical syntax. The following example shows a function **f** being defined that uses **x** as a variable:

```
f(x) = x^2
f, type(f)
|
   (x |--> x^2,
   <class'sage.calculus.calculus.CallableSymbolicExpression'>)
```

Objects created this way are of type CallableSymbolicExpression which means they can be called as shown in the following example:

```
f(4), f(50), f(.2)
|
   (16, 2500, 0.040000000000000010)
```

Here is an example that uses the above CallableSymbolicExpression inside of a loop:

```
a = 0
while a <= 9:
    f(a)
    a = a + 1
|
   0
   1
   4
   9
   16
   25
   36
   49
   64
   81
```

The following example accomplishes the same work that the previous example did, except it uses more advanced language features:

```
a = srange(10)
a
|
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
for num in a:
    f(num)
|
    0
    1
    4
    9
    16
    25
    36
    49
    64
    81
```

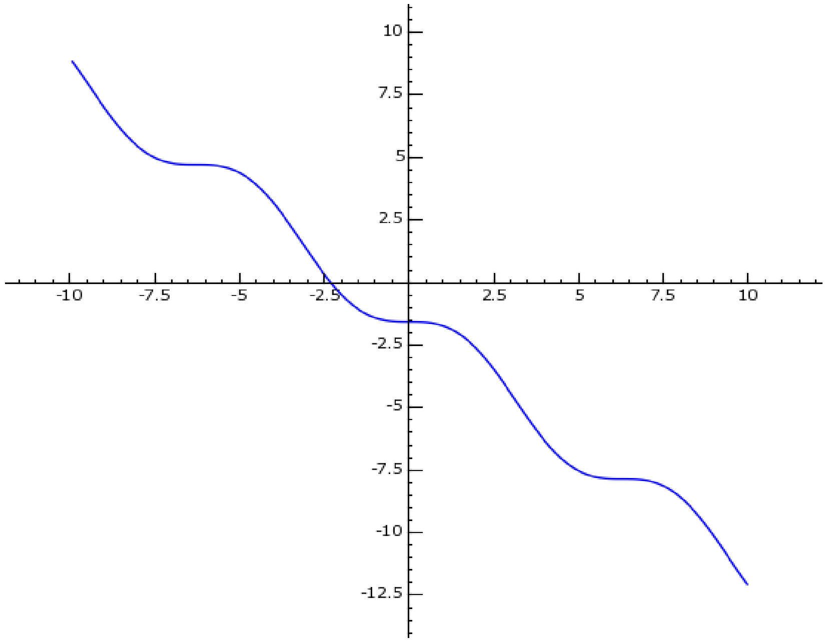
5.14 Finding Roots Graphically And Numerically With The find_root() Method

Sometimes equations cannot be solved algebraically and the solve() function indicates this by returning a copy of the input it was passed. This is shown in the following example:

```
f(x) = sin(x) - x - pi/2
eqn = (f == 0)
solve(eqn, x)
|
    [x == (2*sin(x) - pi)/2]
```

However, equations that cannot be solved algebraically can be solved both graphically and numerically. The following example shows the above equation being solved graphically:

```
show(plot(f,-10,10))
|
```



This graph indicates that the root for this equation is a little greater than -2.5.

The following example shows the equation being solved more precisely using the **find_root()** method:

```
f.find_root(-10,10)

    -2.309881460010057
```

The -10 and +10 that are passed to the **find_root()** method tell it the interval within which it should look for roots.

5.15 Displaying Mathematical Objects In Traditional Form

Earlier it was indicated that SAGE is able to display mathematical objects in either **text form** or **traditional form**. Up until this point, we have been using text form which is the default. If one wants to display a mathematical object in traditional form, the **show()** function can be used. The following example creates a mathematical expression and then displays it in both text form and traditional form:

```
var('y,b,c')
z = (3*y^(2*b))/(4*x^c)^2

#Display the expression in text form.
z
```

```
    3*y^(2*b)/(16*x^(2*c))
```

```
#Display the expression in traditional form.
show(z)
```

$$\frac{3 \cdot y^{2 \cdot b}}{16 \cdot x^{2 \cdot c}}$$

5.15.1 LaTeX Is Used To Display Objects In Traditional Mathematics Form

LaTex (pronounced lā-tek, http://en.wikipedia.org/wiki/LaTeX) is a document markup language which is able to work with a wide range of mathematical symbols. SAGE objects will provide LaTeX descriptions of themselves when their **latex()** methods are called. The LaTeX description of an object can also be obtained by passing it to the **latex()** function:

```
a = (2*x^2)/7
latex(a)
```

```
    \frac{{2 \cdot {x}^{2} }}{7}
```

When this result is fed into LaTeX display software, it will generate traditional mathematics form output similar to the following:

$$\frac{2x^2}{7}$$

The jsMath package which is referenced in Drawing 2.5 is the software that the SAGE Notebook uses to translate LaTeX input into traditional mathematics form output.

5.16 Sets

The following example shows operations that SAGE can perform on sets:

```
a = Set([0,1,2,3,4])
b = Set([5,6,7,8,9,0])
a,b
```

```
    ({0, 1, 2, 3, 4}, {0, 5, 6, 7, 8, 9})
```

```
a.cardinality()
```

```
    5

3 in a
|
    True

3 in b
|
    False

a.union(b)
|
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

a.intersection(b)
|
    {0}
```

6 2D Plotting

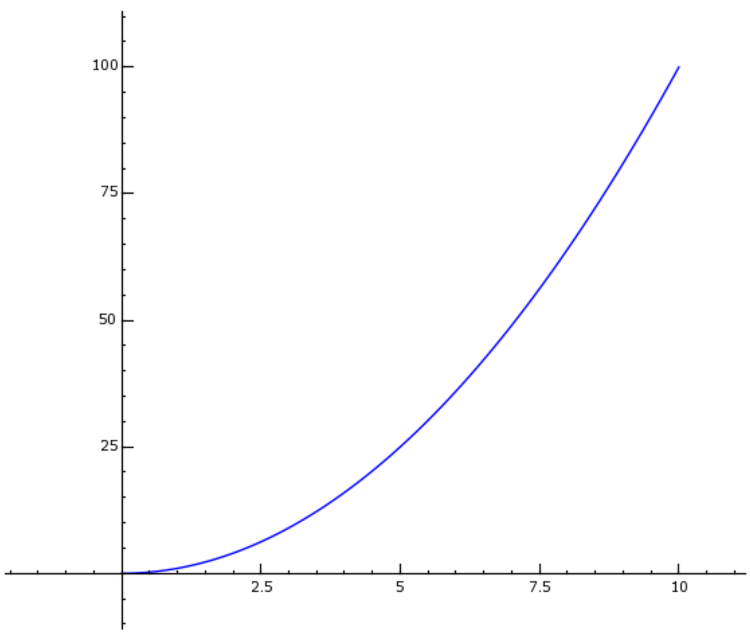
6.1 The plot() And show() Functions

SAGE provides a number of ways to generate 2D plots of mathematical functions and one of these ways is to use the **plot()** function in conjunction with the **show()** function. The following example shows a symbolic expression being passed to the plot() function as its first parameter. The second parameter indicates where plotting should begin on the X axis and the third parameter indicates where plotting should end:

```
a = x^2
b = plot(a, 0, 10)
type(b)

  <class 'sage.plot.plot.Graphics'>
```

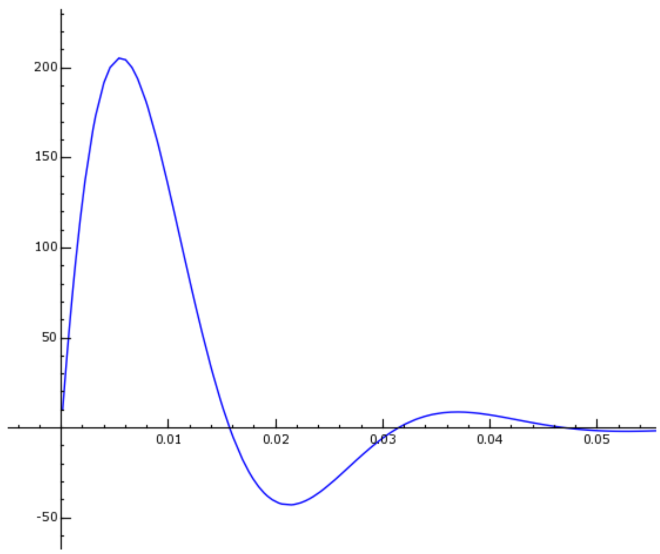
Notice that the **plot()** function does not display the plot. Instead, it creates an object of type sage.plot.plot.Graphics and this object contains the plot data. The **show()** function can then be used to display the plot:

```
show(b)
```



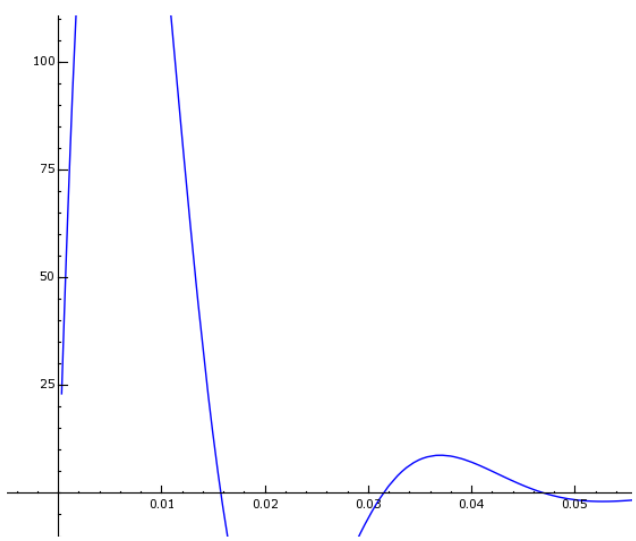
The **show()** function has 4 parameters called **xmin**, **xmax**, **ymin**, and **ymax** that can be used to adjust what part of the plot is displayed. It also has a **figsize** parameter which determines how large the image will be. The following example shows **xmin** and **xmax** being used to display the plot between **0** and **.05** on the **X** axis. Notice that the **plot()** function can be used as the first parameter to the **show()** function in order to save typing effort (Note: if any other symbolic variable other than x is used, it must first be declared with the var() function):

```
v = 400*e^(-100*x)*sin(200*x)
show(plot(v,0,.1),xmin=0, xmax=.05, figsize=[3,3])
```



The **ymin** and **ymax** parameters can be used to adjust how much of the y axis is displayed in the above plot:

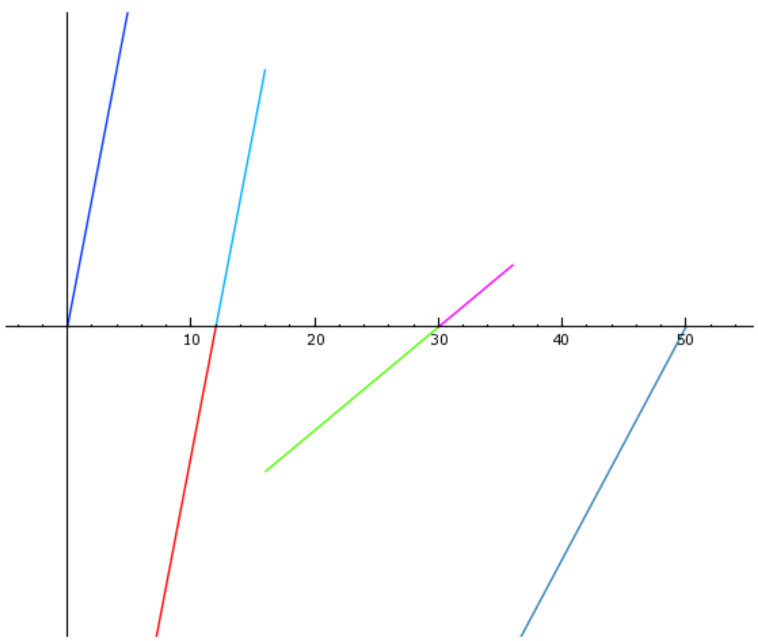
```
show(plot(v,0,.1),xmin=0, xmax=.05, ymin=0, ymax=100, figsize=[3,3])
```



6.1.1 Combining Plots And Changing The Plotting Color

Sometimes it is necessary to combine one or more plots into a single plot. The following example combines 6 plots using the **show()** function:

```
var('t')
p1 = t/4E5
p2 = (5*(t - 8)/2 - 10)/1000000
p3 = (t - 12)/400000
p4 = 0.0000004*(t - 30)
p5 = 0.0000004*(t - 30)
p6 = -0.0000006*(6 - 3*(t - 46)/2)

g1 = plot(p1,0,6,rgbcolor=(0,.2,1))
g2 = plot(p2,6,12,rgbcolor=(1,0,0))
g3 = plot(p3,12,16,rgbcolor=(0,.7,1))
g4 = plot(p4,16,30,rgbcolor=(.3,1,0))
g5 = plot(p5,30,36,rgbcolor=(1,0,1))
g6 = plot(p6,36,50,rgbcolor=(.2,.5,.7))

show(g1+g2+g3+g4+g5+g6,xmin=0, xmax=50, ymin=-.00001, ymax=.00001)
```

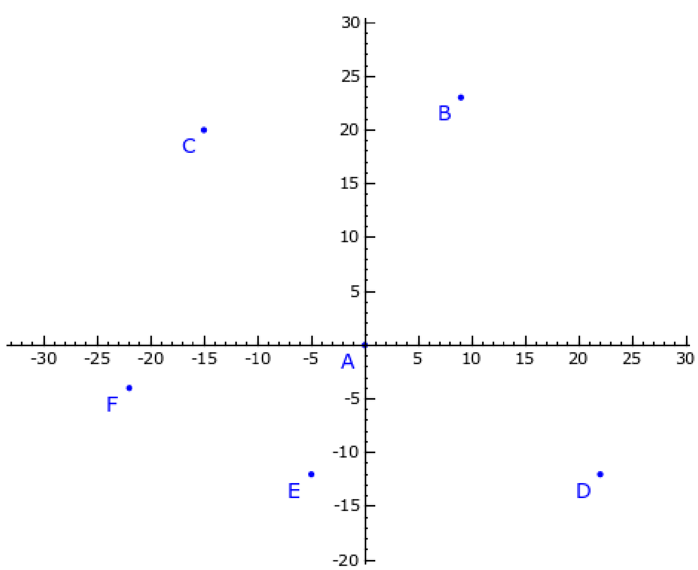


Notice that the color of each plot can be changed using the **rgbcolor** parameter. RGB stands for Red, Green, and Blue and the tuple that is assigned to the **rgbcolor** parameter contains three values between 0 and 1. The first value specifies how much **red** the plot should have (between 0 and 100%), the second value specifies how much **green** the plot should have, and the third value specifies how much **blue** the plot should have.

6.1.2 Combining Graphics With A Graphics Object

It is often useful to combine various kinds of graphics into one image. In the following example, 6 points are plotted along with a text label for each plot:

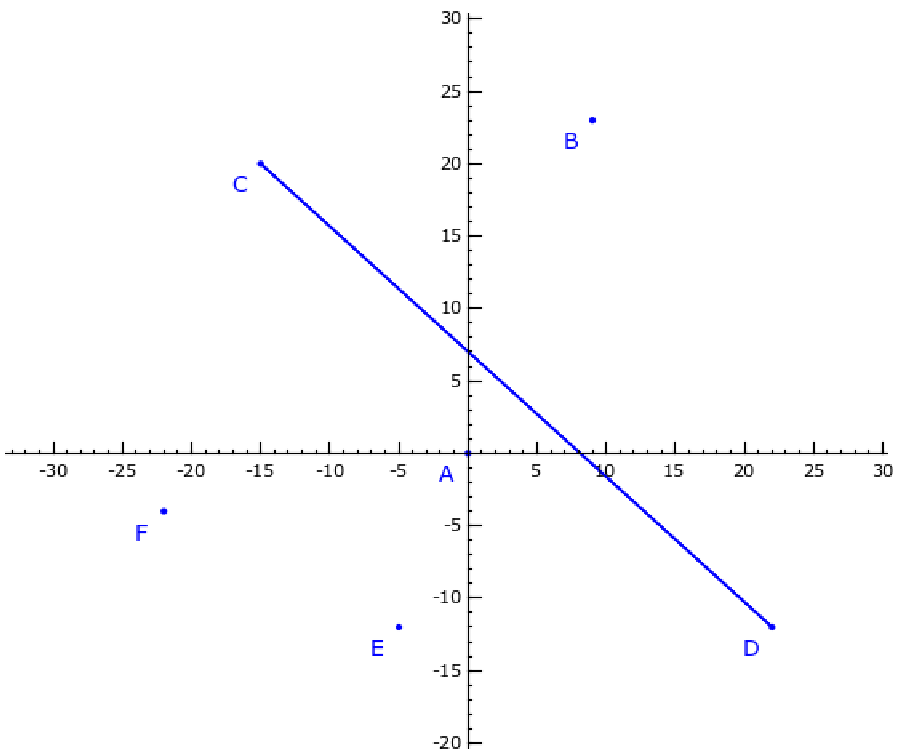
```
"""
Plot the following points on a graph:

A (0,0)
B (9,23)
C (-15,20)
D (22,-12)
E (-5,-12)
F (-22,-4)
"""

#Create a Graphics object which will be used to hold multiple
# graphics objects.  These graphics objects will be displayed
# on the same image.
g = Graphics()

#Create a list of points and add them to the graphics object.
points=[(0,0), (9,23), (-15,20), (22,-12), (-5,-12), (-22,-4)]
g += point(points)

#Add labels for the points to the graphics object.
for (pnt,letter) in zip(points,['A','B','C','D','E','F']):
    g += text(letter,(pnt[0]-1.5, pnt[1]-1.5))

#Display the combined graphics objects.
show(g,figsize=[5,4])
```



First, an empty Graphics object is instantiated and a list of plotted points are created using the point() function. These plotted points are then added to the Graphics object using the += operator. Next, a label for each point is added to the Graphics object using a **for** loop. Finally, the Graphics object is displayed in the worksheet using the show() function.

Even after being displayed, the Graphics object still contains all of the graphics that have been placed into it and more graphics can be added to it as needed. For example, if a line needed to be drawn between points C and D, the following code can be executed in a separate cell to accomplish this:

```
g += line([(-15,20), (22,-12)])
show(g)
```

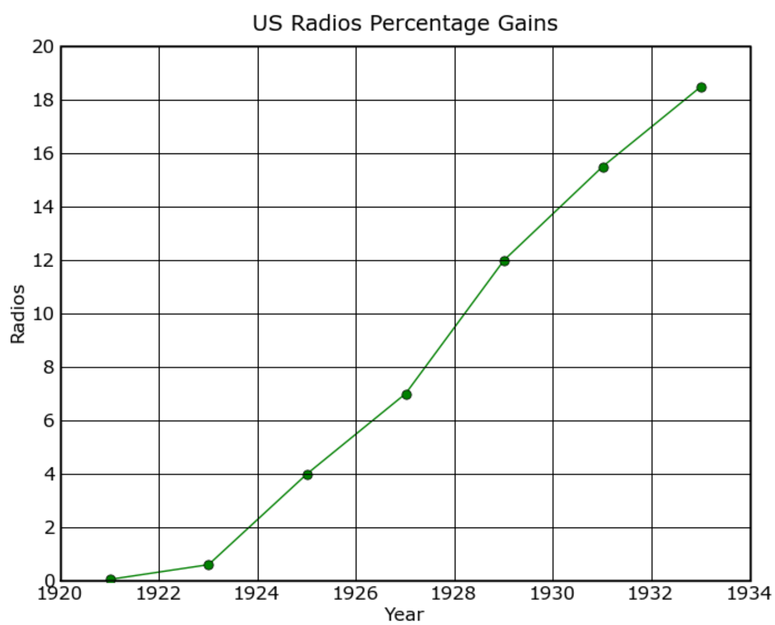


6.2 Advanced Plotting With matplotlib

SAGE uses the **matplotlib** (http://matplotlib.sourceforge.net) library for its plotting needs and if one requires more control over plotting than the **plot()** function provides, the capabilities of matplotlib can be used directly. While a complete explanation of how matplotlib works is beyond the scope of this book, this section provides examples that should help you to begin using it.

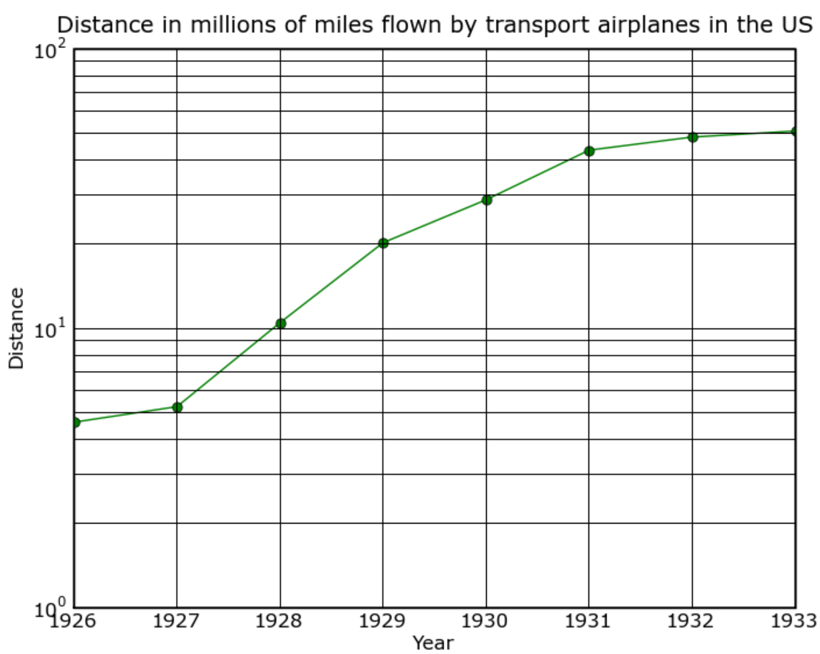
6.2.1 Plotting Data From Lists With Grid Lines And Axes Labels

```
x = [1921, 1923, 1925, 1927, 1929, 1931, 1933]
y = [ .05, .6, 4.0, 7.0, 12.0, 15.5, 18.5]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('US Radios Percentage Gains')
ax.set_xlabel('Year')
ax.set_ylabel('Radios')
ax.plot(x,y, 'go-', linewidth=1.0 )
canvas.print_figure('ex1_linear.png')

```



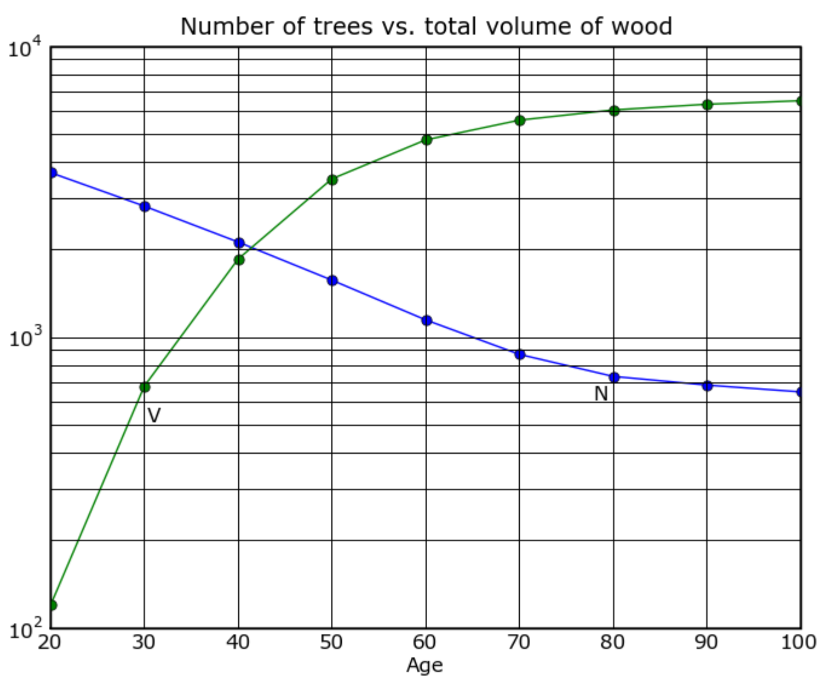
6.2.2 Plotting With A Logarithmic Y Axis

```
x = [1926, 1927, 1928, 1929, 1930, 1931, 1932, 1933]
y = [ 4.61,5.24, 10.47, 20.24, 28.83, 43.40, 48.34, 50.80]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('Distance in millions of miles flown by transport
airplanes in the US')
ax.set_xlabel('Year')
ax.set_ylabel('Distance')
ax.semilogy(x,y, 'go-', linewidth=1.0 )
canvas.print_figure('ex2_log.png')

```



6.2.3 Two Plots With Labels Inside Of The Plot

```
x = [20,30,40,50,60,70,80,90,100]
y = [3690,2830,2130,1575,1150,875,735,686,650]
z = [120,680,1860,3510,4780,5590,6060,6340,6520]

from matplotlib.backends.backend_agg import FigureCanvasAgg as \
FigureCanvas
from matplotlib.figure import Figure
from matplotlib.ticker import *
from matplotlib.dates import *
fig = Figure()
canvas = FigureCanvas(fig)
ax = fig.add_subplot(111)
ax.xaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.set_major_locator( MaxNLocator(10) )
ax.yaxis.set_major_formatter( FormatStrFormatter( '%d' ))
ax.yaxis.grid(True, linestyle='-', which='minor')
ax.grid(True, linestyle='-', linewidth=.5)
ax.set_title('Number of trees vs. total volume of wood')
ax.set_xlabel('Age')
ax.set_ylabel('')
ax.semilogy(x,y, 'bo-', linewidth=1.0 )
ax.semilogy(x,z, 'go-', linewidth=1.0 )
ax.annotate('N', xy=(550, 248),  xycoords='figure pixels')
ax.annotate('V', xy=(180, 230),  xycoords='figure pixels')
canvas.print_figure('ex5_log.png')
```



7 SAGE Usage Styles

SAGE is an extremely flexible environment and therefore there are multiple ways to use it. In this chapter, two SAGE usage styles are discussed and they are called the **Speed** style and the **OpenOffice Presentation** style.

The Speed usage style is designed to solve problems as quickly as possible by minimizing the amount of effort that is devoted to making results look good. This style has been found to be especially useful for solving end of chapter problems that are usually present in mathematics related textbooks.

The OpenOffice Presentation style is designed to allow a person with no mathematical document creation skills to develop mathematical documents with minimal effort. This presentation style is useful for creating homework submissions, reports, articles, books, etc. and this book was developed using this style.

7.1 The Speed Usage Style

(In development...)

7.2 The OpenOffice Presentation Usage Style

(In development...)

8 High School Math Problems (most of the problems are still in development)

8.1 Pre-Algebra

Wikipedia entry.	http://en.wikipedia.org/wiki/Pre-algebra

(In development...)

8.1.1 Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/Equation

(In development...)

8.1.2 Expressions

Wikipedia entry.	http://en.wikipedia.org/wiki/Mathematical_expression

(In development...)

8.1.3 Geometry

Wikipedia entry.	http://en.wikipedia.org/wiki/Geometry

(In development...)

8.1.4 Inequalities

Wikipedia entry.	http://en.wikipedia.org/wiki/Inequality

(In development...)

8.1.5 Linear Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Linear_functions

(In development...)

8.1.6 Measurement

Wikipedia entry.	http://en.wikipedia.org/wiki/Measurement

(In development...)

8.1.7 Nonlinear Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Nonlinear_system

(In development...)

8.1.8 Number Sense And Operations

Wikipedia entry.	http://en.wikipedia.org/wiki/Number_sense
Wikipedia entry.	http://en.wikipedia.org/wiki/Operation_(mathematics)

(In development...)

8.1.8.1 Express an integer fraction in lowest terms

```
"""
Problem:
Express 90/105 in lowest terms.

Solution:
One way to solve this problem is to factor both the numerator and the
denominator into prime factors, find the common factors, and then
divide both the numerator and denominator by these factors.
"""
n = 90
d = 105
print n,n.factor()
print d,d.factor()
|
   Numerator: 2 * 3^2 * 5
   Denominator: 3 * 5 * 7


"""
It can be seen that the factors 3 and 5 each appear once in both the
numerator and denominator, so we divide both the numerator and
denominator by 3*5:
"""
n2 = n/(3*5)
d2 = d/(3*5)
print "Numerator2:",n2
print "Denominator2:",d2
|
   Numerator2: 6
   Denominator2: 7


"""
Therefore, 6/7 is 90/105 expressed in lowest terms.
```

```
This problem could also have been solved more directly by simply
entering 90/105 into a cell because rational number objects are
automatically reduced to lowest terms:
"""
90/105

    6/7
```

8.1.9 Polynomial Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Polynomial_function

(In development...)

8.2 Algebra

Wikipedia entry.	http://en.wikipedia.org/wiki/Algebra_1

(In development...)

8.2.1 Absolute Value Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Absolute_value

(In development...)

8.2.2 Complex Numbers

Wikipedia entry.	http://en.wikipedia.org/wiki/Complex_numbers

(In development...)

8.2.3 Composite Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Composite_function

(In development...)

8.2.4 Conics

Wikipedia entry.	http://en.wikipedia.org/wiki/Conics

(In development...)

8.2.5 Data Analysis

Wikipedia entry.	http://en.wikipedia.org/wiki/Data_analysis

(In development...)

9 Discrete Mathematics: Elementary Number And Graph Theory

Wikipedia entry.	http://en.wikipedia.org/wiki/Discrete_mathematics

(In development...)

9.1.1 Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/Equation

(In development...)

9.1.1.1 Express a symbolic fraction in lowest terms

```
"""
Problem:
Express (6*x^2 - b) / (b - 6*a*b) in lowest terms, where a and b
represent positive integers.

Solution:
"""

var('a,b')
n = 6*a^2 - a
d = b - 6 * a * b
print n
print "                                          ---------"
print d
|
                                                  2
                                               6 a  - a
                                               ---------
                                               b - 6 a b


"""
We begin by factoring both the numerator and the denominator and then
looking for common factors:
"""
n2 = n.factor()
d2 = d.factor()
print "Factored numerator:",n2.__repr__()
print "Factored denominator:",d2.__repr__()
|
```

```
    Factored numerator: a*(6*a - 1)
    Factored denominator: -(6*a - 1)*b
```

```
"""
At first, it does not appear that the numerator and denominator
contain any common factors.  If the denominator is studied further,
however, it can be seen that if (1 - 6 a) is multiplied by -1,
(6 a - 1) is the result and this factor is also present
in the numerator.  Therefore, our next step is to multiply both the
numerator and denominator by -1:
"""
n3 = n2 * -1
d3 = d2 * -1
print "Numerator * -1:",n3.__repr__()
print "Denominator * -1:",d3.__repr__()
|
    Numerator * -1: -a*(6*a - 1)
    Denominator * -1: (6*a - 1)*b
```

```
"""
Now, both the numerator and denominator can be divided by (6*a - 1)
in order to reduce each to lowest terms:
"""
common_factor = 6*a - 1
n4 = n3 / common_factor
d4 = d3 / common_factor
print n4
print "                                      ---"
print d4
|
                                      - a
                                      ---
                                       b
```

```
"""
The problem could also have been solved more directly using a
SymbolicArithmetic object:
"""
z = n/d
z.simplify_rational()
|
    -a/b
```

9.1.1.2 Determine the product of two symbolic fractions

Perform the indicated operation: $(\frac{x}{2y})^2 . (\frac{4y^2}{3x})^3$

```
"""
Since symbolic expressions are usually automatically simplified, all
that needs to be done with this problem is to enter the expression
and assign it to a variable:
"""

var('y')
a = (x/(2*y))^2 * ((4*y^2)/(3*x))^3

#Display the expression in text form:
a

   16*y^4/(27*x)


#Display the expression in traditional form:
show(a)
```

$$\frac{16 \cdot y^4}{27 \cdot x}$$

9.1.1.3 Solve a linear equation for x

Solve $3x+2x-8=5x-3x+7$

```
"""
Like terms will automatically be combined when this equation is
placed into a SymbolicEquation object:
"""
a = 5*x + 2*x - 8 == 5*x - 3*x + 7
a

   7*x - 8 == 2*x + 7
```

"""
First, lets move the x terms to the left side of the equation by subtracting 2x from each side. (Note: remember that the underscore '_' holds the result of the last cell that was executed:
"""

```
_ - 2*x
|
    5*x - 8 == 7
```

"""
Next, add 8 to both sides:
"""

```
_+8
|
    5*x == 15
```

"""
Finally, divide both sides by 5 to determine the solution:
"""

```
_/5
|
    x == 3
```

"""
This problem could also have been solved automatically using the solve() function:
"""

```
solve(a,x)
|
    [x == 3]
```

9.1.1.4 Solve a linear equation which has fractions

Solve $\frac{16x-13}{6}=\frac{3x+5}{2}-\frac{4-x}{3}$

```
"""
The first step is to place the equation into a SymbolicEquation
object.  It is good idea to then display the equation so that you can
verify that it was entered correctly:
"""
a = (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3
a
|
    (16*x - 13)/6 == (3*x + 5)/2 - (4 - x)/3

"""
In this case, it is difficult to see if this equation has been
entered correctly when it is displayed in text form so lets also
display it in traditional form:
```

```
"""
show(a)
```

$$\frac{16\cdot x-13}{6} = \frac{3\cdot x+5}{2} - \frac{4-x}{3}$$

```
"""
The next step is to determine the least common denominator (LCD) of
the fractions in this equation so the fractions can be removed:
"""
lcm([6,2,3])

    6

"""
The LCD of this equation is 6 so multiplying it by 6 removes the
fractions:
"""
b = a*6
b

    16*x - 13 == 6*((3*x + 5)/2 - (4 - x)/3)

"""
The right side of this equation is still in factored form so expand
it:
"""
c = b.expand()
c

    16*x - 13 == 11*x + 7

"""
Transpose the 11x to the left side of the equals sign by subtracting
11x from the SymbolicEquation:
"""
d = c - 11*x
d

    5*x - 13 == 7

"""
Transpose the -13 to the right side of the equals sign by adding 13
to the SymbolicEquation:
"""
e = d + 13
e
```

```
|
    5*x == 20

"""
Finally, dividing the SymbolicEquation by 5 will leave x by itself on
the left side of the equals sign and produce the solution:
"""
f = e / 5
f
|
    x == 4

"""
This problem could have also be solved automatically using the
solve() function:
"""
solve(a,x)
|
    [x == 4]
```

9.1.2 Exponential Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Exponential_function

(In development...)

9.1.3 Exponents

Wikipedia entry.	http://en.wikipedia.org/wiki/Exponent

(In development...)

9.1.4 Expressions

Wikipedia entry.	http://en.wikipedia.org/wiki/Expression_(mathematics)

(In development...)

9.1.5 Inequalities

Wikipedia entry.	http://en.wikipedia.org/wiki/Inequality

(In development...)

9.1.6 Inverse Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Inverse_function

2726 (In development...)

9.1.7 Linear Equations And Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Linear_functions

2727 (In development...)

9.1.8 Linear Programming

Wikipedia entry.	http://en.wikipedia.org/wiki/Linear_programming

2728 (In development...)

9.1.9 Logarithmic Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Logarithmic_function

2729 (In development...)

9.1.10 Logistic Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Logistic_function

2730 (In development...)

9.1.11 Matrices

Wikipedia entry.	http://en.wikipedia.org/wiki/Matrix_(mathematics)

2731 (In development...)

9.1.12 Parametric Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/Parametric_equation

2732 (In development...)

9.1.13 Piecewise Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Piecewise_function

2733 (In development...)

9.1.14 Polynomial Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Polynomial_function

2734 (In development...)

9.1.15 Power Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Power_function

2735 (In development...)

9.1.16 Quadratic Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Quadratic_function

2736 (In development...)

9.1.17 Radical Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Nth_root

2737 (In development...)

9.1.18 Rational Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Rational_function

2738 (In development...)

9.1.19 Sequences

Wikipedia entry.	http://en.wikipedia.org/wiki/Sequence

2739 (In development...)

9.1.20 Series

Wikipedia entry.	http://en.wikipedia.org/wiki/Series_mathematics

2740 (In development...)

9.1.21 Systems of Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/System_of_equations

2741 (In development...)

9.1.22 Transformations

Wikipedia entry.	http://en.wikipedia.org/wiki/Transformation_(geometry)

2742 (In development...)

9.1.23 Trigonometric Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Trigonometric_function

2743 (In development...)

2744 9.2 Precalculus And Trigonometry

Wikipedia entry.	http://en.wikipedia.org/wiki/Precalculus
	http://en.wikipedia.org/wiki/Trigonometry

2745 (In development...)

9.2.1 Binomial Theorem

Wikipedia entry.	http://en.wikipedia.org/wiki/Binomial_theorem

2746 (In development...)

9.2.2 Complex Numbers

Wikipedia entry.	http://en.wikipedia.org/wiki/Complex_numbers

2747 (In development...)

9.2.3 Composite Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Composite_function

2748 (In development...)

9.2.4 Conics

Wikipedia entry.	http://en.wikipedia.org/wiki/Conics

2749 (In development...)

9.2.5 Data Analysis

Wikipedia entry.	http://en.wikipedia.org/wiki/Data_analysis

2750 (In development...)

10 Discrete Mathematics: Elementary Number And Graph Theory

Wikipedia entry.	http://en.wikipedia.org/wiki/Discrete_mathematics

2751 (In development...)

10.1.1 Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/Equation

2752 (In development...)

10.1.2 Exponential Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Equation

2753 (In development...)

10.1.3 Inverse Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Inverse_function

2754 (In development...)

10.1.4 Logarithmic Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Logarithmic_function

2755 (In development...)

10.1.5 Logistic Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Logistic_function

2756 (In development...)

10.1.6 Matrices And Matrix Algebra

Wikipedia entry.	http://en.wikipedia.org/wiki/Matrix_(mathematics)

2757 (In development...)

10.1.7 Mathematical Analysis

Wikipedia entry.	http://en.wikipedia.org/wiki/Mathematical_analysis

2758 (In development...)

10.1.8 Parametric Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/Parametric_equation

2759 (In development...)

10.1.9 Piecewise Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Piecewise_function

2760 (In development...)

10.1.10 Polar Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/Polar_equation

2761 (In development...)

10.1.11 Polynomial Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Polynomial_function

2762 (In development...)

10.1.12 Power Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Power_function

2763 (In development...)

10.1.13 Quadratic Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Quadratic_function

2764 (In development...)

10.1.14 Radical Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Nth_root

2765 (In development...)

10.1.15 Rational Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Rational_function

2766 (In development...)

10.1.16 Real Numbers

Wikipedia entry.	http://en.wikipedia.org/wiki/Real_number

2767 (In development...)

10.1.17 Sequences

Wikipedia entry.	http://en.wikipedia.org/wiki/Sequence

2768 (In development...)

10.1.18 Series

Wikipedia entry.	http://en.wikipedia.org/wiki/Series_(mathematics)

2769 (In development...)

10.1.19 Sets

Wikipedia entry.	http://en.wikipedia.org/wiki/Set

2770 (In development...)

10.1.20 Systems of Equations

Wikipedia entry.	http://en.wikipedia.org/wiki/System_of_equations

2771 (In development...)

10.1.21 Transformations

Wikipedia entry.	http://en.wikipedia.org/wiki/Transformation_(geometry)

2772 (In development...)

10.1.22 Trigonometric Functions

Wikipedia entry.	http://en.wikipedia.org/wiki/Trigonometric_function

2773 (In development...)

10.1.23 Vectors

Wikipedia entry.	http://en.wikipedia.org/wiki/Vector

2774 (In development...)

2775 10.2 Calculus

Wikipedia entry.	http://en.wikipedia.org/wiki/Calculus

2776 (In development...)

10.2.1 Derivatives

Wikipedia entry.	http://en.wikipedia.org/wiki/Derivative

2777 (In development...)

10.2.2 Integrals

Wikipedia entry.	http://en.wikipedia.org/wiki/Integral

2778 (In development...)

10.2.3 Limits

Wikipedia entry.	http://en.wikipedia.org/wiki/Limit_(mathematics)

(In development...)

10.2.4 Polynomial Approximations And Series

Wikipedia entry.	http://en.wikipedia.org/wiki/Convergent_series

(In development...)

10.3 Statistics

Wikipedia entry.	http://en.wikipedia.org/wiki/Statistics

(In development...)

10.3.1 Data Analysis

Wikipedia entry.	http://en.wikipedia.org/wiki/Data_analysis

(In development...)

10.3.2 Inferential Statistics

Wikipedia entry.	http://en.wikipedia.org/wiki/Inferential_statistics

(In development...)

10.3.3 Normal Distributions

Wikipedia entry.	http://en.wikipedia.org/wiki/Normal_distribution

(In development...)

10.3.4 One Variable Analysis

Wikipedia entry.	http://en.wikipedia.org/wiki/Univariate

(In development...)

10.3.5 Probability And Simulation

Wikipedia entry.	http://en.wikipedia.org/wiki/Probability

(In development...)

10.3.6 Two Variable Analysis

Wikipedia entry.	http://en.wikipedia.org/wiki/Multivariate

(In development...)

2789 11 High School Science Problems

2790 (In development...)

2791 11.1 Physics

Wikipedia entry.	http://en.wikipedia.org/wiki/Physics

2792 (In development...)

11.1.1 Atomic Physics

Wikipedia entry.	http://en.wikipedia.org/wiki/Atomic_physics

2793 (In development...)

11.1.2 Circular Motion

Wikipedia entry.	http://en.wikipedia.org/wiki/Circular_motion

2794 (In development...)

11.1.3 Dynamics

Wikipedia entry.	http://en.wikipedia.org/wiki/Dynamics_(physics)

2795 (In development...)

11.1.4 Electricity And Magnetism

Wikipedia entry.	http://en.wikipedia.org/wiki/Electricity
	http://en.wikipedia.org/wiki/Magnetism

2796 (In development...)

11.1.5 Fluids

Wikipedia entry.	http://en.wikipedia.org/wiki/Fluids

2797 (In development...)

11.1.6 Kinematics

Wikipedia entry.	http://en.wikipedia.org/wiki/Kinematics

2798 (In development...)

11.1.7 Light

Wikipedia entry.	http://en.wikipedia.org/wiki/Light

2799 (In development...)

11.1.8 Optics

Wikipedia entry.	http://en.wikipedia.org/wiki/Optics

2800 (In development...)

11.1.9 Relativity

Wikipedia entry.	http://en.wikipedia.org/wiki/Relativity

2801 (In development...)

11.1.10 Rotational Motion

Wikipedia entry.	http://en.wikipedia.org/wiki/Rotational_motion

2802 (In development...)

11.1.11 Sound

Wikipedia entry.	http://en.wikipedia.org/wiki/Sound

2803 (In development...)

11.1.12 Waves

Wikipedia entry.	http://en.wikipedia.org/wiki/Waves

2804 (In development...)

11.1.13 Thermodynamics

Wikipedia entry.	http://en.wikipedia.org/wiki/Thermodynamics

2805 (In development...)

11.1.14 Work

Wikipedia entry.	http://en.wikipedia.org/wiki/Mechanical_work

2806 (In development...)

11.1.15 Energy

Wikipedia entry.	http://en.wikipedia.org/wiki/Energy

2807 (In development...)

11.1.16 Momentum

Wikipedia entry.	http://en.wikipedia.org/wiki/Momentum

2808 (In development...)

11.1.17 Boiling

Wikipedia entry.	http://en.wikipedia.org/wiki/Boiling

2809 (In development...)

11.1.18 Buoyancy

Wikipedia entry.	http://en.wikipedia.org/wiki/Bouyancy

2810 (In development...)

11.1.19 Convection

Wikipedia entry.	http://en.wikipedia.org/wiki/Convection

2811 (In development...)

11.1.20 Density

Wikipedia entry.	http://en.wikipedia.org/wiki/Density

2812 (In development...)

11.1.21 Diffusion

Wikipedia entry.	http://en.wikipedia.org/wiki/Diffusion

2813 (In development...)

11.1.22 Freezing

Wikipedia entry.	http://en.wikipedia.org/wiki/Freezing

2814 (In development...)

11.1.23 Friction

Wikipedia entry.	http://en.wikipedia.org/wiki/Friction

2815 (In development...)

11.1.24 Heat Transfer

Wikipedia entry.	http://en.wikipedia.org/wiki/Heat_transfer

2816 (In development...)

11.1.25 Insulation

Wikipedia entry.	http://en.wikipedia.org/wiki/Insulation

2817 (In development...)

11.1.26 Newton's Laws

Wikipedia entry.	http://en.wikipedia.org/wiki/Newtons_laws

2818 (In development...)

11.1.27 Pressure

Wikipedia entry.	http://en.wikipedia.org/wiki/Pressure

2819 (In development...)

11.1.28 Pulleys

Wikipedia entry.	http://en.wikipedia.org/wiki/Pulley

2820 (In development...)

12 Fundamentals Of Computation

12.1 What Is A Computer?

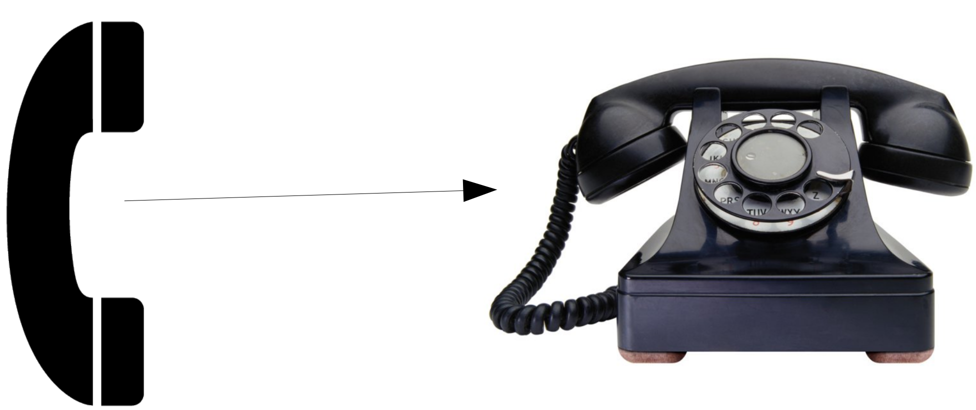
Many people think computers are difficult to understand because they are complex. Computers are indeed complex, but this is not why they are difficult to understand. Computers are difficult to understand because only a small part of a computer exists in the physical world. The physical part of a computer is the only part a human can see and the rest of a computer exists in a nonphysical world which is invisible. This invisible world is the **world of ideas** and most of a computer exists as ideas in this nonphysical world.

The key to understanding computers is to understand that the purpose of these idea-based machines is to automatically manipulate ideas of all types. The name 'computer' is not very helpful for describing what computers really are and perhaps a better name for them would be Idea Manipulation Devices or IMDs.

Since ideas are nonphysical objects, they cannot be brought into the physical world and neither can physical objects be brought into the world of ideas. Since these two worlds are separate from each other, the only way that physical objects can manipulate objects in the world of ideas is through remote control via symbols.

12.2 What Is A Symbol?

A **symbol** is an object that is used to represent another object. Drawing 12.1 shows an example of a symbol of a telephone which is used to represent a physical telephone.

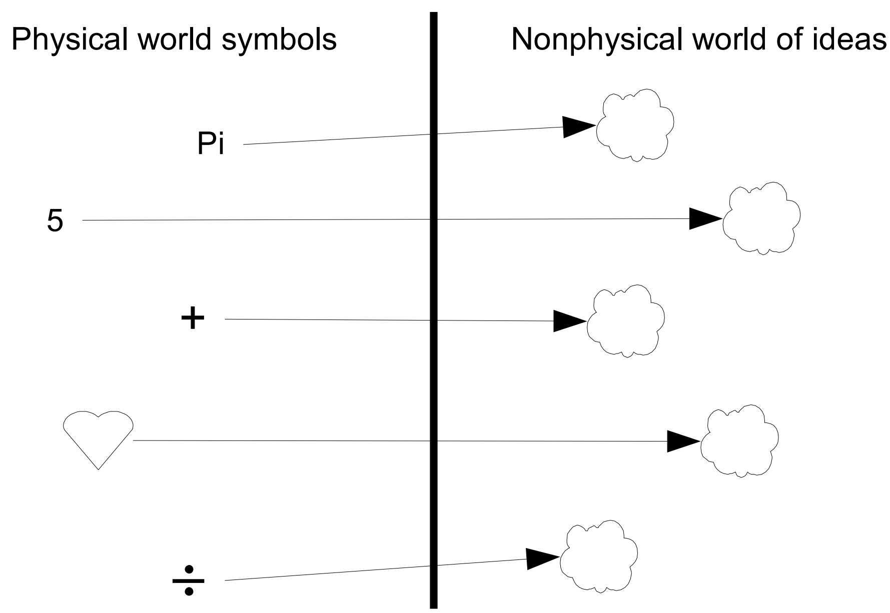


Drawing 12.1: Symbol associated with a physical object.

The symbol of a telephone shown in Drawing 12.1 is usually created with ink printed on a flat surface (like a piece of paper). In general, though, any type of physical matter (or property of physical matter) that is arranged into a **pattern** can be used as a symbol.

12.3 Computers Use Bit Patterns As Symbols

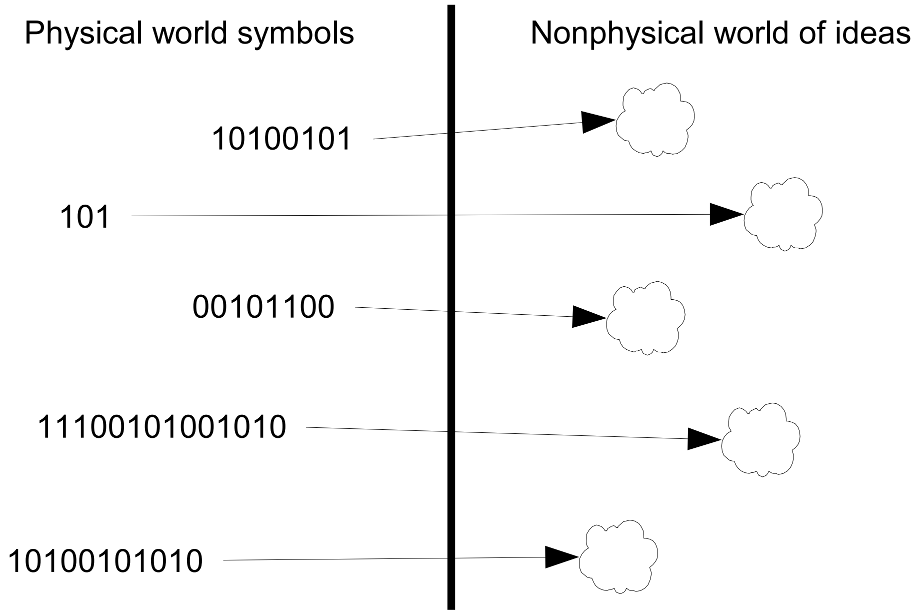
Symbols which are made of physical matter can represent all types of physical objects, but they can also be used to represent nonphysical objects in the world of ideas. (see Drawing 12.2)



Drawing 12.2: Physical symbols can represent nonphysical ideas.

Among the simplest symbols that can be formed out of physical matter are bits and patterns of bits. A single bit can only be placed into two states which are the **on** state and the **off** state. When written, typed, or drawn, a bit in the **on** state is represented by the numeral **1** and when it is in the **off** state it is represented by the numeral **0**. Patterns of bits look like the following when they are written, typed, or drawn: 101, 100101101, 0101001100101, 10010.

Drawing 12.3 shows how bit patterns can be used just as easily as any other symbols made of physical matter to represent nonphysical ideas.

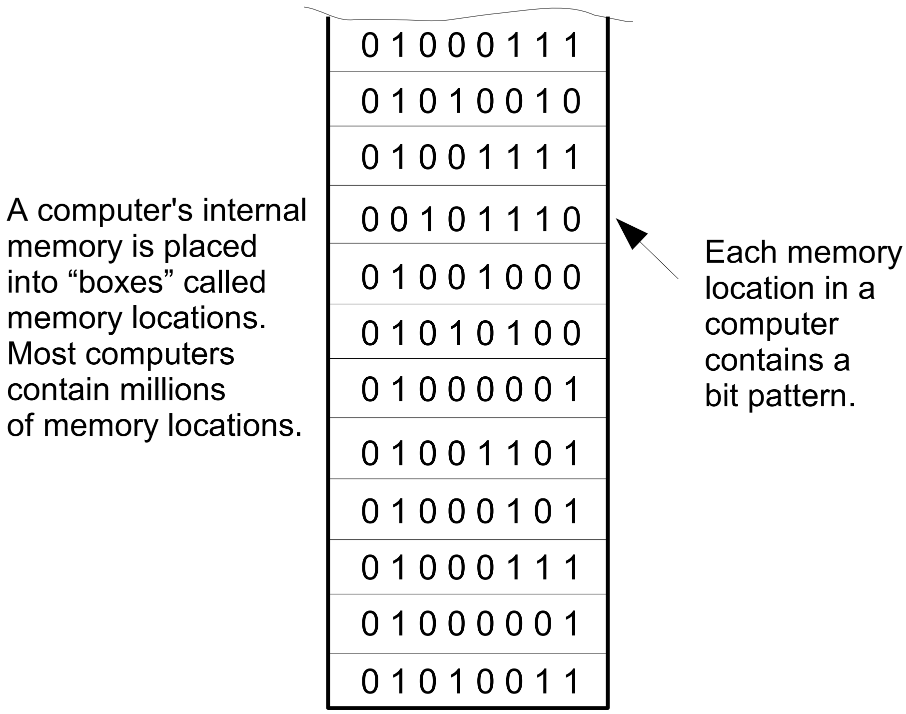


Drawing 12.3: Bits can also represent nonphysical ideas.

Other methods for forming physical matter into bits and bit patterns include: varying the tone of an audio signal between two frequencies, turning a light on and off, placing or removing a magnetic field on the surface of an object, and changing the voltage level between two levels in an electronic device. Most computers use the last method to hold bit patterns that represent ideas.

A computer's internal memory consists of numerous "boxes" called **memory locations** and **each memory location contains a bit pattern that can be used to represent an idea**. Most computers contain millions of memory locations which allow them to easily reference millions of ideas at the same time. Larger computers contain billions of memory locations. For example, a typical personal computer purchased in 2007 contains over 1 billion memory locations.

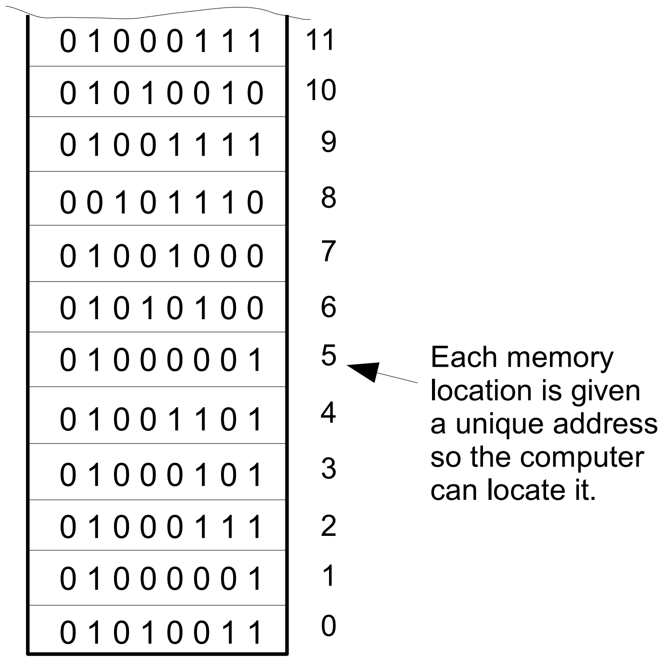
Drawing 12.4 shows a section of the internal memory of a small computer along with the bit patterns that this memory contains.



Drawing 12.4: Computer memory locations contain bit patterns.

Each of the millions of bit pattern symbols in a computer's internal memory are capable of representing any idea a human can think of. The large number of bit patterns that most computers contain, however, would be difficult to keep track of without the use of some kind of organizing system.

The system that computers use to keep track of the many bit patterns they contain consists of giving each memory location a unique address as shown in Drawing 12.5.



Drawing 12.5: Each memory location is given a unique address.

12.4 Contextual Meaning

At this point you may be wondering "how one can determine what the bit patterns in a memory location, or a set of memory locations, mean?" The answer to this question is that a concept called **contextual meaning** gives bit patterns their meaning.

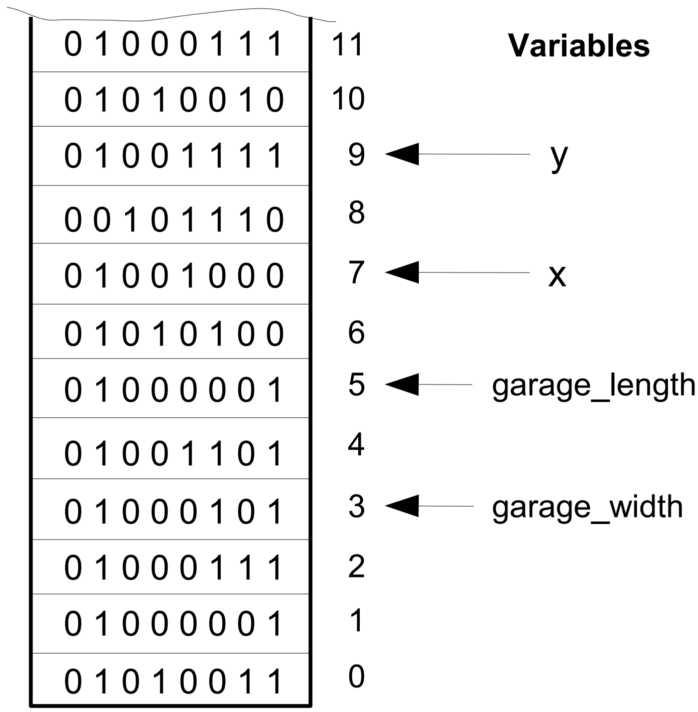
Context is the circumstances within which an event happens or the environment within which something is placed. **Contextual meaning**, therefore, is the meaning that a context gives to the events or things that are placed within it.

Most people use contextual meaning every day, but they are not aware of it. Contextual meaning is a very powerful concept and it is what enables a computer's memory locations to reference any idea that a human can think of. Each memory location can hold a bit pattern, but a human can have that bit pattern mean anything they wish. If more bits are needed to hold a given pattern than are present in a single memory location, the pattern can be spread across more than one location.

12.5 Variables

Computers are very good at remembering numbers and this allows them to keep track of numerous addresses with ease. Humans, however, are not nearly as good at remembering numbers as computers are and so a concept called a **variable** was invented to solve this problem.

A variable is a **name** that can be associated with a memory address so that humans can refer to bit pattern symbols in memory using a **name** instead of a **number**. Drawing 12.6 shows four variables that have been associated with 4 memory addresses inside of a computer.



Drawing 12.6: Using variables instead of memory addresses.

The variable names **garage_width** and **garage_length** are referencing memory locations that hold patterns that represent the dimensions of a garage and the variable names **x** and **y** are referencing memory locations that might represent numbers in an equation. Even though this description of the above variables is accurate, it is fairly tedious to use and therefore most of the time people just say or write something like "the variable garage_length holds the length of the garage."

A variable is used to symbolically represent an attribute of an object. Even though a typical personal computer is capable of holding millions of variables, most objects possess a greater number of attributes than the capacity of most computers can hold. For example, a 1 kilogram rock contains approximately 10,000,000,000,000,000,000,000,000 atoms. [1] Representing even just the positions of this rock's atoms is currently well beyond the capacity of even the most advanced computer. Therefore, computers usually work with models of

1 "The Singularity Is Near" Ray Kurzweil, Viking.

objects instead of complete representations of them.

12.6 Models

A **model** is a simplified representation of an object that only references some of its attributes. Examples of typical object attributes include weight, height, strength, and color. The attributes that are selected for modeling are chosen for a given purpose. The more attributes that are represented in the model, the more expensive the model is to make. Therefore, only those attributes that are absolutely needed to achieve a given purpose are usually represented in a model. The process of selecting only some of an object's attributes when developing a model of it is called **abstraction**.

The following is an example which illustrates the process of problem solving using models. Suppose we wanted to build a garage that could hold 2 cars along with a workbench, a set of storage shelves, and a riding lawn mower. Assuming that the garage will have an adequate ceiling height, and that we do not want to build the garage any larger than it needs to be for our stated purpose, how could an adequate length and width be determined for the garage?

One strategy for determining the size of the garage is to build perhaps 10 garages of various sizes in a large field. When the garages are finished, take 2 cars to the field along with a workbench, a set of storage shelves, and a riding lawn mower. Then, place these items into each garage in turn to see which is the smallest one that these items will fit into without being too cramped.

The test garages in the field can then be discarded and a garage which is the same size as the one that was chosen could be built at the desired location. Unfortunately, 11 garages would need to be built using this strategy instead of just one and this would be very expensive and inefficient.

A way to solve this problem less expensively is by using a **model of the garage** and **models of the items that will be placed inside it**. Since we only want to determine the dimensions of the garage's floor, we can make a scaled down model of just its floor using a piece of paper.

Each of the items that will be placed into the garage could also be represented by scaled-down pieces of paper. Then, the pieces of paper that represent the items can be placed on top of the the large piece of paper that represents the floor and these smaller pieces of paper can be moved around to see how they fit. If the items are too cramped, a larger piece of paper can be cut to represent the floor and, if the items have too much room, a smaller piece of paper for the floor can be cut.

When a good fit is found, the length and width of the piece of paper that represents the floor can be measured and then these measurements can be scaled up to the units used for the full-size garage. With this method, only a few pieces of paper are needed to solve the problem instead of 10 full-size garages that will later be discarded.

The only attributes of the full-sized objects that were copied to the pieces of paper were the object's length and width. As this example shows, paper models are significantly easier to work with than the objects they represent. However, **computer variables are even easier to use for modeling than paper or almost any other kind of modeling mechanism**.

At this point, though, the paper-based modeling technique has one important advantage over the computer variables we have look at. The paper model was able to be changed by moving the item models around and changing the size of the paper garage floor. The variables we have discussed so have been given the ability to represent an object attribute, but no mechanism has been given yet that would allow the variable's to change. A computer without the ability to change the contents of its variables would be practically useless.

12.7 Machine Language

Earlier is was stated that bit patterns in a computer's memory locations can be used to represent any ideas that a human can think of. If memory locations can represent any idea, this means that they can reference ideas that represent **instructions** which tell a computer how to automatically manipulate the variables in its memory.

The part of a computer that follows the instructions that are in its memory is called a Central Processing Unit (CPU) or a microprocessor. When a microprocessor is following instructions in its memory, it is also said to be **running** them or **executing** them.

Microprocessors are categorized into families and each microprocessor family has its own set of instructions (called an **instruction set**) that is different than the instructions that other microprocessor family's use. A microprocessor's instruction set represents the building blocks of a language that can be used to tell it what to do. This language is formed by placing **sequences of instructions from the instruction set into memory** and it the only language that a microprocessor is able to understand. Since this is the only language a microprocessor is able to understand, it is called **machine language**. A sequence of machine language instructions is called a **computer program** and a person who creates sequences of machine language instructions in order to tell the computer what to do is called a **programmer**.

We will now look at what the instruction set of a simple microprocessor looks like along with a simple program which has been developed using this instruction set.

Here is the instruction set for the 6500 family of microprocessors:

```
ADC  ADd memory to accumulator with Carry.
AND  AND memory with accumulator.
ASL  Arithmetic Shift Left one bit.
BCC  Branch on Carry Clear.
BCS  Branch on Carry Set.
BEQ  Branch on result EQual to zero.
BIT  test BITs in accumulator with memory.
BMI  Branch on result MInus.
BNE  Branch on result Not Equal to zero.
BPL  Branch on result PLus).
BRK  force Break.
BVC  Branch on oVerflow flag Clear.
BVS  Branch on oVerflow flag Set.
CLC  CLear Carry flag.
CLD  CLear Decimal mode.
CLI  CLear Interrupt disable flag.
CLV  CLear oVerflow flag.
CMP  CoMPare memory and accumulator.
CPX  ComPare memory and index X.
CPY  ComPare memory and index Y.
DEC  DECrement memory by one.
DEX  DEcrement register S by one.
DEY  DEcrement register Y by one.
EOR  Exclusive OR memory with accumulator.
INC  INCrement memory by one.
INX  INcrement register X by one.
INY  INcrement register Y by one.
JMP  JuMP to new memory location.
JSR  Jump to SubRoutine.
LDA  LoaD Accumulator from memory.
LDX  LoaD X register from memory.
LDY  LoaD Y register from memory.
LSR  Logical Shift Right one bit.
NOP  No OPeration.
ORA  OR memory with Accumulator.
PHA  PusH Accumulator on stack.
PHP  PusH Processor status on stack.
PLA  PuLl Accumulator from stack.
PLP  PuLl Processor status from stack.
ROL  ROtate Left one bit.
ROR  ROtate Right one bit.
RTI  ReTurn from Interrupt.
RTS  ReTurn from Subroutine.
SBC  SuBtract with Carry.
```

```
SEC  SEt Carry flag.
SED  SEt Decimal mode.
SEI  SEt Interrupt disable flag.
STA  STore Accumulator in memory.
STX  STore Register X in memory.
STY  STore Register Y in memory.
TAX  Transfer Accumulator to register X.
TAY  Transfer Accumulator to register Y.
TSX  Transfer Stack pointer to register X.
TXA  Transfer register X to Accumulator.
TXS  Transfer register X to Stack pointer.
TYA  Transfer register Y to Accumulator.
```

The following is a small program which has been written using the 6500 family's instruction set. The purpose of the program is to calculate the sum of the 10 numbers which have been placed into memory started at address 0200 hexadecimal.

Here are the 10 numbers in memory (which are printed in blue) along with the memory location that the sum will be stored into (which is printed in red). 0200 here is the address in memory of the first number.

```
0200  01 02 03 04 05 06 07 08 - 09 0A 00 00 00 00 00
00  ................

```

Here is a program that will calculate the sum of these 10 numbers:

```
0250  A2 00     LDX #00h
0252  A9 00     LDA #00h
0254  18        CLC
0255  7D 00 02  ADC 0200h,X
0258  E8        INX
0259  E0 0A     CPX #0Ah
025B  D0 F8     BNE 0255h
025D  8D 0A 02  STA 020Ah
0260  00        BRK
...
```

After the program was executed, the sum it calculated was stored in memory. The sum was determined to be 37 hex (which is 55 decimal) and it is shown here printed in red:

```
0200  01 02 03 04 05 06 07 08 - 09 0A 37 00 00 00 00 00  ..........
7.....
```

Of course, you are not expected to understand how this assembly language program works. The purpose for showing it to you is so you can see what a program that uses a microprocessor's instruction set looks like.

Low Level Languages And High Level Languages

Even though programmers are able to program a computer using the instructions in its instruction set, this is a tedious task. The early computer programmers wanted to develop programs in a language that was more like a natural language, English for example, than the machine language that microprocessors understand. Machine language is considered to be a **low level languages** because it was designed to be simple so that it could be easily executed by the circuits in a microprocessor.

Programmers then figured out ways to use low level languages to create the **high level languages** that they wanted to program in. This is when languages like FORTRAN (in 1957), ALGOL (in 1958), LISP (in 1959), COBOL (in 1960), BASIC (in 1964) and C (1972) were created. Ultimately, a microprocessor is only capable of understanding machine language and therefore all programs that are written in a high level language must be converted into machine language before they can be executed by a microprocessor.

The rules that indicate how to properly type in code for a given programming language are called **syntax** rules. If a programmer does not follow the language's syntax rules when typing in a program, the software that transforms the source code into machine language will become confused and then issue what is called a syntax error.

As an example of what a syntax error might look like, consider the word 'print'. If the word 'print' was a command in a given program language, and the programmer typed 'pvint' instead of 'print', this would be a syntax error.

12.8 Compilers And Interpreters

There are two types of programs that are commonly used to convert a higher level language into machine language. The first kind of program is called a **compiler** and it takes a high-level language's **source code** (which is usually in typed form) as its input and converts it into machine language. After the machine language equivalent of the source code has been generated, it can be loaded into a computer's memory and executed. The compiled version of a program can also be saved on a storage device and loaded into a computer's memory whenever it is needed.

The second type of program that is commonly used to convert a high-level language into machine language is called an **interpreter**. Instead of converting source code into machine language like a compiler does, an interpreter reads the source code (usually one line at a time), determines what actions this line of source code is suppose to accomplish, and then it performs these actions. It then looks at the next line of source code underneath the one it just finished interpreting, it determines what actions this next line of code wants done, it performs these actions, and so on.

Thousands of computer languages have been created since the 1940's, but there are currently around 2 to 3 hundred historically important languages. Here is a link to a website that lists a number of the historically important computer languages: http://en.wikipedia.org/wiki/Timeline_of_programming_languages

12.9 Algorithms

A computer programmer certainly needs to know at least one programming language, but when a programmer solves a problem, they do it at a level that is higher in abstraction than even the more abstract computer languages.

After the problem is solved, then the solution is encoded into a programming language. It is almost as if a programmer is actually two people. The first person is the **problem solver** and the second person is the **coder**.

For simpler problems, many programmers create algorithms in their minds and encode these algorithm directly into a programming language. They switch back and forth between being the problem solver and the coder during this process.

With more complex programs, however, the problem solving phase and the coding phase are more distinct. The algorithm which solves a given problem is is developed using means other than a programming language and then it is recored in a document. This document is then passed from the **problem solver** to the **coder** for encoding into a programming language.

The first thing that a problem solver will do with a problem is to **analyze** it. This is an extremely important step because if a problem is not analyzed, then it can not be properly solved. To **analyze** something means to break it down into its component parts and then these parts are studied to determine how they work. A well known saying is '**divide and conquer**' and when a difficult problem is analyzed, it is broken down into smaller problems which are each simpler to solve than the overall problem. The **problem solver** then develops an **algorithm** to solve each of the simpler problems and, when these algorithms are combined, they form the solution to the overall problem.

An **algorithm** (pronounced al-gor-rhythm) is a sequence of instructions which describe how to accomplish a given task. These instructions can be expressed in various ways including writing them in natural languages (like English), drawing **diagrams** of them, and encoding them in a programming language.

The concept of an algorithm came from the various procedures that mathematicians developed for solving mathematical problems, like calculating the sum of 2 numbers or calculating their product.

Algorithms can also be used to solve more general problems. For example, the following algorithm could have been followed by a person who wanted to solve the garage sizing problem using paper models:

1) Measure the length and width of each item that will be placed into the garage using metric units and record these measurements.

2) Divide the measurements from step 1 by 100 then cut out pieces of paper that match these dimensions to serve as models of the original items.

3) Cut out a piece of paper which is 1.5 times as long as the model of the largest car and 3 times wider than it to serve as a model of the garage floor.

4) Locate where the garage doors will be placed on the model of the garage floor, mark the locations with a pencil, and place the models of both cars on top of the model of the garage floor, just within the perimeter of the paper and between the two pencil marks.

5) Place the models of the items on top of the model of the garage floor in the empty space that is not being occupied by the models of the cars.

6) Move the models of the items into various positions within this empty space to determine how well all the items will fit within this size garage.

7) If the fit is acceptable, go to step 10.

8) If there is not enough room in the garage, increase the length dimension, the width dimension (or both dimensions) of the garage floor model by 10%, create a new garage floor model, and go to step 4.

9) If there is too much room in the garage, decrease the length dimension, the width dimension (or both dimensions) of the garage model by 10%, create a new garage floor model, and go to step 4.

10) Measure the length and width dimensions of the garage floor model, multiply these dimensions by 100, and then build the garage using these larger dimensions.

As can be seen with this example, an algorithm often contains a significant number of steps because it needs to be detailed enough so that it leads to the desired solution. After the steps have been developed and recorded in a document, however, they can be followed over and over again by people who need to solve the given problem.

12.10 Computation

It is fairly easy to understand how a human is able to follow the steps of an algorithm, but it is more difficult to understand how computer can perform these steps when its microprocessor is only capable of executing simple machine language instructions.

In order to understand how a microprocessor is able to perform the steps in an algorithm, one must first understand what **computation** (which is also known as **calculation**) is. Lets search for some good definitions of each of these words on the Internet and read what they have to say.”

Here are two definitions for the word **computation**:

1) The manipulation of numbers or symbols according to fixed rules. Usually applied to the operations of an automatic electronic computer, but by extension to some processes performed by minds or brains. (www.informatics.susx.ac.uk/books/computers-and-thought/gloss/node1.html)

2) A computation can be seen as a purely physical phenomenon occurring inside a closed physical system called a computer. Examples of such physical systems include digital computers, quantum computers, DNA computers, molecular computers, analog computers or wetware computers. (www.informatics.susx.ac.uk/books/computers-and-thought/gloss/node1.html)

These two definitions indicate that **computation** is the "**manipulation of numbers or symbols according to fixed rules"** and that it "**can be seen as a purely physical phenomenon occurring inside a closed physical system called a computer**." Both definitions indicate that the machines we normally think of as computers are just **one type of computer** and that other types of closed physical systems can also act as computers. These other types of computers include DNA computers, molecular computers, analog computers, and wetware computers (or brains).

The following two definitions for **calculation** shed light on the kind of rules that normal computers, brains, and other types of computers use:

1) A calculation is a deliberate process for transforming one or more inputs into one or more results. (en.wikipedia.org/wiki/Calculation)

2) Calculation: the procedure of calculating; determining something by mathematical or logical methods (wordnet.princeton.edu/perl/webwn)

These definitions for calculation indicate that it "**is a deliberate process for transforming one or more inputs into one or more results"** and that this is done "**by mathematical or logical methods"**. We do not yet completely understand what mathematical and logical methods brains use to perform calculations, but rapid progress is being made in this area.

The second definition for calculation uses the word **logic** and this word needs to be defined before we can proceed:

> The logic of a system is the whole structure of rules that must be used for any reasoning within that system. Most of mathematics is based upon a well-understood structure of rules and is considered to be highly logical. It is always necessary to state, or otherwise have it understood, what rules are being used before any logic can be applied. (ddi.cs.uni-potsdam.de/Lehre/TuringLectures/MathNotions.htm)

Reasoning is the process of using predefined rules to move from one point in a system to another point in the system. For example, when a person adds 2 numbers together on a piece of paper, they must follow the rules of the addition algorithm in order to obtain a correct sum. The addition algorithm's rules are its **logic** and, when someone applies these rules during a calculation, they are **reasoning** with the rules.

Lets now apply these concepts to the question about how a computer can perform the steps of an algorithm when its microprocessor is only capable of executing simple machine language instructions. When a person develops an algorithm, the steps in the algorithm are usually stated as high-level tasks which do not contain all of the smaller steps that are necessary to perform each task.

For example, a person might write a step that states "Drive from New York to San Francisco." This large step can be broken down into smaller steps that contain instructions such as "turn left at the intersection, go west for 10 kilometers, etc." If all of the smaller steps in a larger step are completed, then the larger step is completed too.

A human that needs to perform this large driving step would usually be able to figure out what smaller steps need to be performed in order accomplish it. Computers are extremely stupid, however, and before any algorithm can be executed on a computer, the algorithm's steps must be broken down into smaller steps, and these smaller steps must be broken down into even small steps, until the steps are simple enough to be performed by the instruction set of a microprocessor.

Sometimes only a few smaller steps are needed to implement a larger step, but sometimes hundreds or even thousands of smaller steps are required. Hundreds or thousands of smaller steps will translate into hundreds or thousands of machine language instructions when the algorithm is converted into machine language.

If machine language was the only language that computers could be programmed in, then most algorithms would be too large to be placed into a computer by a human. An algorithm that is encoded into a high-level language, however, does not need to be broken down into as many smaller steps as would be needed with machine language. The hard work of further breaking down an algorithm that has been encoded into a high-level language is automatically done by either a compiler or an interpreter. This is why most of the time, programmers use a high-level language to develop in instead of machine language.

12.11 Diagrams Can Be Used To Record Algorithms

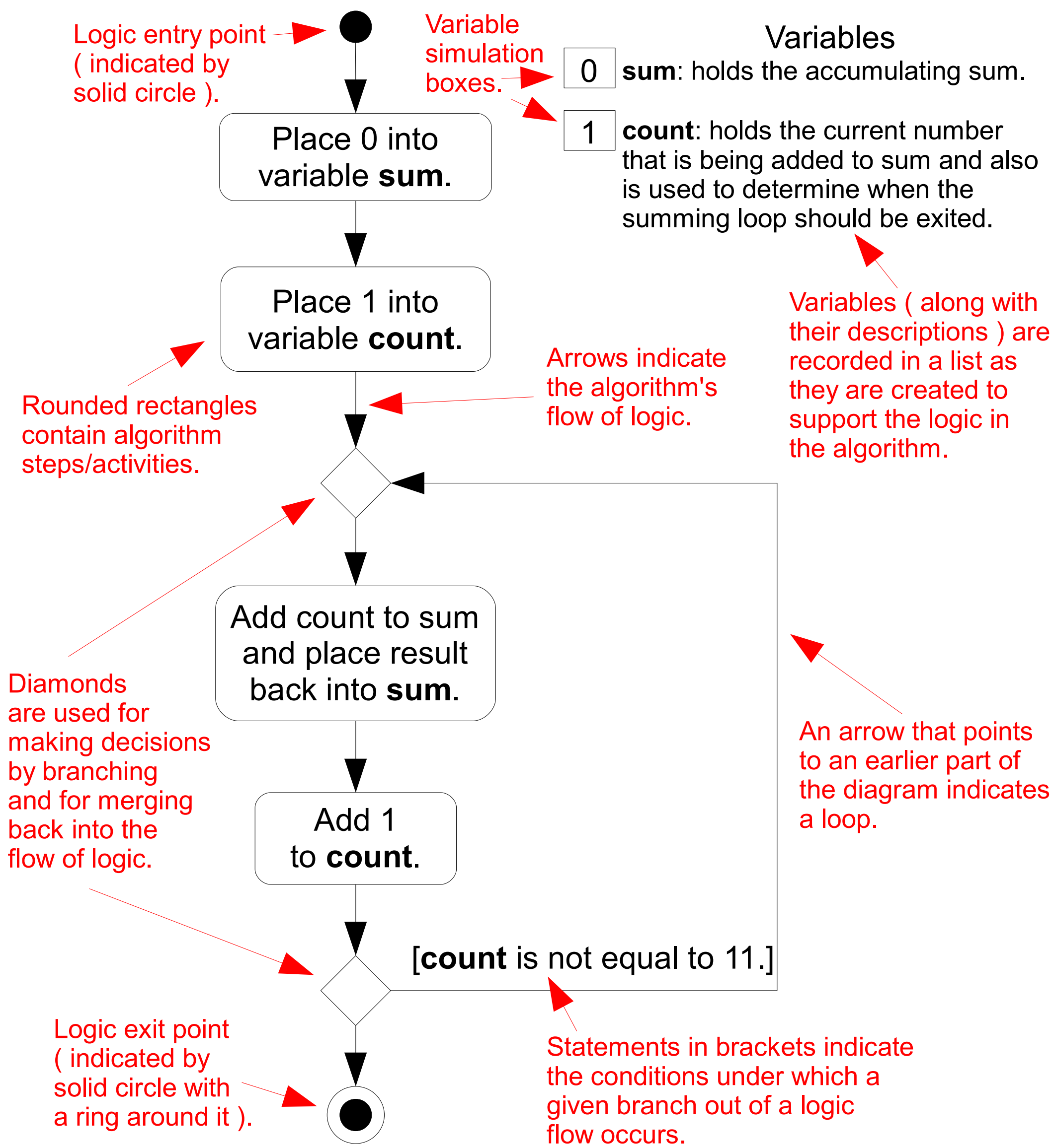
Earlier it was mentioned that not only can an algorithm can be recorded in a natural language like English but it can also be recorded using diagrams. You may be surprised to learn, however, that a whole diagram-based language has been created which allows all aspects of a program to be designed by 'problem solvers', including the algorithms that a program uses. This language is call **UML** which stands for **Unified Modeling Language**. One of UML's diagrams is called an **Activity diagram** and it can be used to show the sequence of steps (or activities) that are part of some piece of logic. The following is an example which shows how an algorithm can be represented in an Activity diagram.

12.12 Calculating The Sum Of The Numbers Between 1 And 10

The first thing that needs to be done with a problem before it can be analyzed and solved is to describe it clearly and accurately. Here is a short description for the problem we will solve with an algorithm:

Description: In this problem, the sum of the numbers between 1 and 10 inclusive needs to be determined.

Inclusive here means that the numbers 1 and 10 will be included in the sum. Since this is a fairly simple problem we will not need to spend too much time analyzing it. Drawing 12.7 shows an algorithm for solving this problem that has been placed into an Activity diagram.



Drawing 12.7: Activity diagram for an algorithm.

An algorithms and its Activity diagram are developed at the same time. During the development process, variables are created as needed and their names are usually recorded in a list along with their descriptions. The developer periodically starts at the entry point and walks through the logic to make sure it is correct. Simulation boxes are placed next to each variable so that they can be use to record and update how the logic is changing the variable's values. During a walk-through, errors are usually found and these need to be fixed by moving flow arrows and adjusting the text that is inside of the activity rectangles.

When the point where no more errors in the logic can be found, the developer can stop being the **problem solver** and pass the algorithm over to the **coder** so it can be encoded into a programming language.

12.13 The Mathematics Part Of Mathematics Computing Systems

Mathematics has been described as the "science of patterns" [2]. Here is a definition for pattern:

```
1) Systematic arrangement...
 (http://www.answers.com/topic/pattern)
```

And here is a definition for system:

```
1) A group of interacting, interrelated, or interdependent elements
forming a complex whole.

2) An organized set of interrelated ideas or principles.
(http://www.answers.com/topic/system)
```

Therefore, mathematics can be though of as a science that deals with the systematic properties of physical and nonphysical objects. The reason that mathematics is so powerful is that all physical and nonphysical objects posses systematic properties and therefore, mathematics is a means by which these objects can be understood and manipulated.

The more mathematics a person knows, the more control they are able to have over the physical world. This makes mathematics one of the most useful and exciting areas of knowledge a person can possess.

Traditionally, learning mathematics also required learning the numerous tedious and complex algorithms that were needed to perform written calculations with mathematics. Usually over 50% of the content of the typical traditional math textbook is devoted to teaching writing-based algorithms and an even higher percentage of the time a person spends working through a textbook is spent manually working these algorithms.

For most people, learning and performing tedious, complex written-calculation algorithms is so difficult and mind-numbingly boring that they never get a chance to see that the "mathematics" part of mathematics is extremely exciting, powerful, and beautiful.

The bad news is that writing-based calculation algorithms will always be tedious, complex, and boring. The good news is that the invention of mathematics computing environments has significantly reduced the need for people to use writing-based calculation algorithms.

2 Steen, Lynn Arthur. “The Science of Patterns.” *Science* 240 (April 1988): 611-616.

13 Setting Up A SAGE Server

As indicated in a previous section, most people will first use SAGE as a web service and the assumption was made at the beginning of this book that the reader already had access to a SAGE server. This section is for people who want to have their own SAGE server and it covers obtaining, installing, configuring, and maintaining one on Windows or Linux.

Since the SAGE Notebook Server is based on Internet technologies, this section will start by covering some of these technologies. A high-level view of SAGE's architecture will then be given followed by a discussion of the contents of the SAGE distribution files. Finally, setting up both Linux and Windows-based SAGE servers will be covered.

13.1 An Introduction To Internet-based Technologies

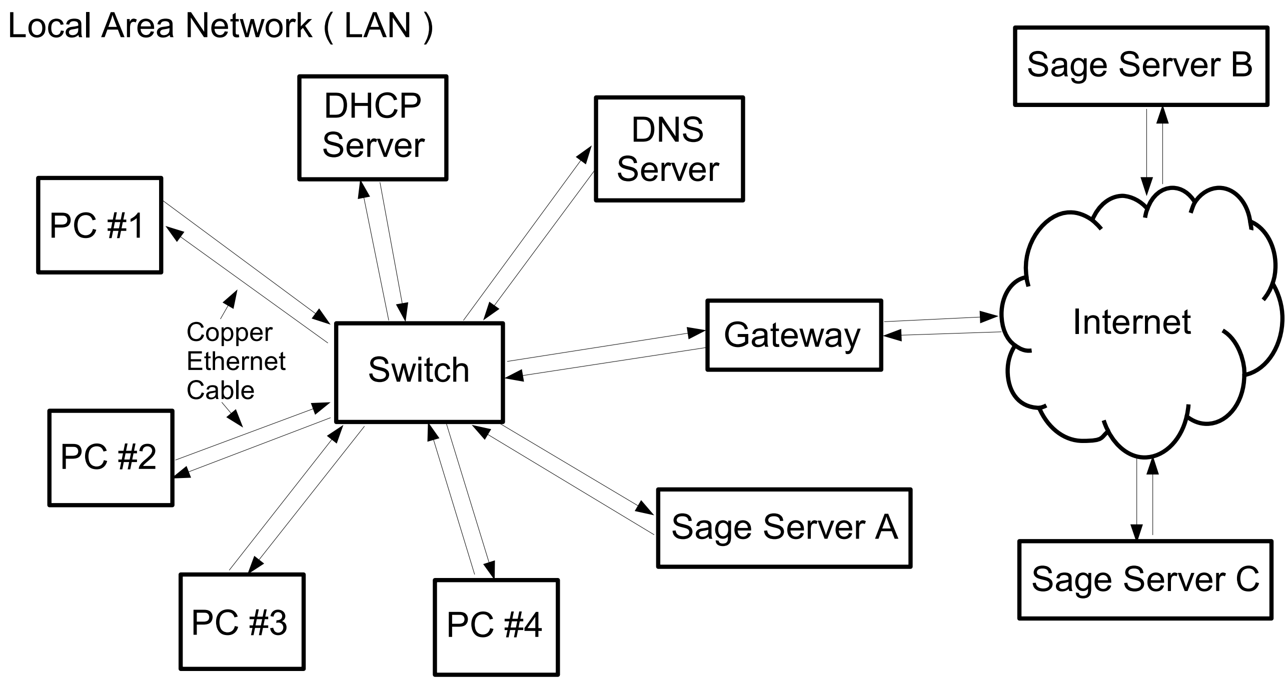
The Internet is currently one of the most important technologies of our civilization and its importance will only increase in the future. In fact, the Internet is expanding so quickly that projections show almost all computing devices will eventually be connected to it (https://embeddedjava.dev.java.net/resources/waves_of_the_internet_telemetry.pdf). Therefore, understanding how Internet-related technologies work is valuable for anyone who is interested in working with computers.

Understanding the history of how the Internet was created is also valuable, but we will not be discussing this history here because it has been well documented elsewhere. I highly recommend that you do an Internet search on the history of the Internet and read some of the articles you find. I assure you that it will be an excellent investment of your time.

13.1.1 How do multiple computers communicate with each other?

When only 2 computers need to communicate with each other, the situation is simple because all that is needed is to connect them together with a communications medium (such as copper wires, fiber optic cables, or wireless radio signals). The information that leaves one computer is sent to the other computer and vice versa. But what about the situation where multiple computers need to communication with each other? There are a number of ways to solve this problem and one of the more common ways is shown in Figure 11:

Figure 11 shows multiple computers connected to what is called a **Local Area Network** or **LAN**. A **LAN** consists of multiple computers that are physically close to each other (usually in the same room or in the same building) and attached to each other using some kind of communications medium. In Drawing 13.1, the computers are attached to a device called a **switch** with copper Ethernet cables.



Drawing 13.1: A Local Area Network (LAN)

Computers on a network communicate with each other using **messages** and sending a message is similar to sending a letter through the mail. The purpose of a **switch** is to look at each message that is sent into it, determine which computer the message is being sent to, and then sending the message to that computer.

There is a problem with the model in Figure 11, however, because the names that are associated with each computer on the network would not be suitable for uniquely identifying them if their numbers would be increased into the hundreds or thousands. Beyond this, the cloud on the right side of the figure represents the Internet and the millions of computers (which are also called **hosts**) that are currently attached to it. Messages can also be sent to these computers and received from them, but only if each computer on the Internet is uniquely identified in some way. Beyond this, rules for how the messages are to be exchanged must also exist.

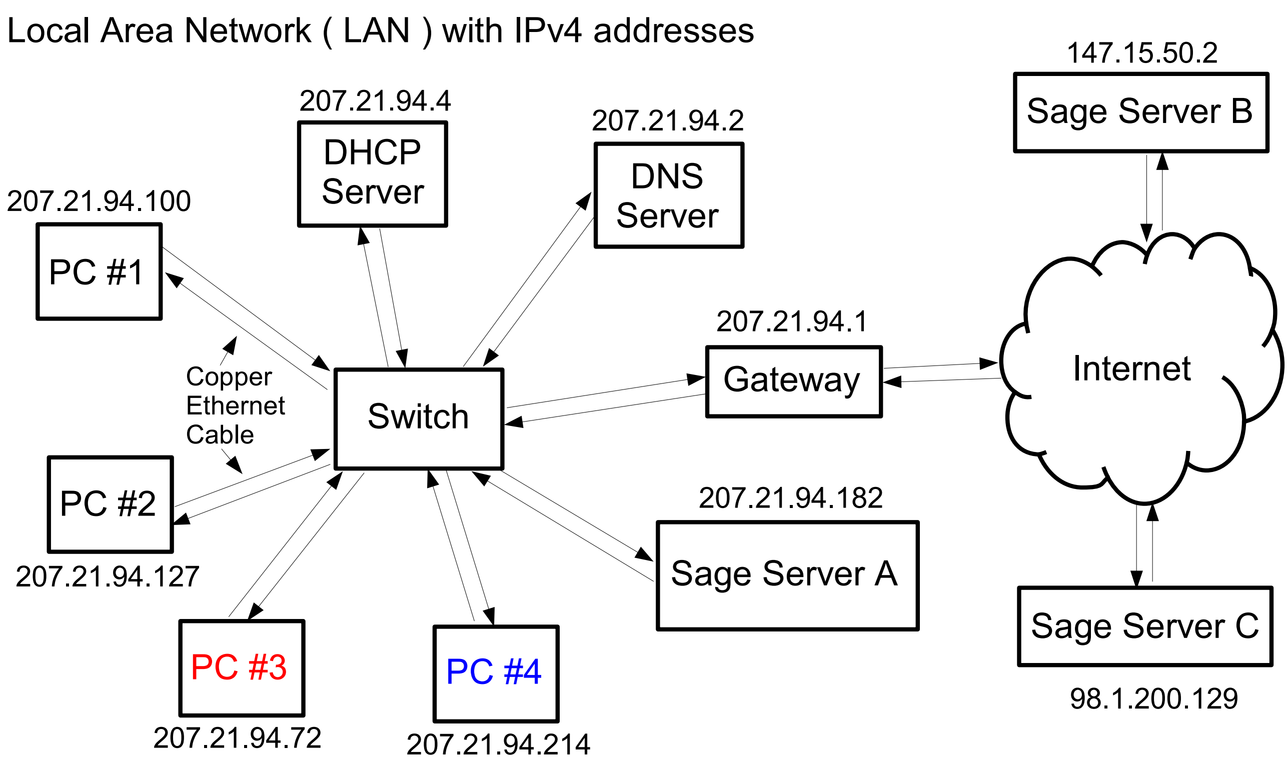
13.1.2 The TCP/IP protocol suite

Two problems that needed to be solved before the Internet could be created were 1) each computer needed to be uniquely identified and 2) communications rules (also called **protocols**) needed to be developed which determined how the messages were to be exchanged. With respect to the Internet, a **protocol** can be defined as "**a set of rules that define an exact format for communication between systems.**" (www.unitedyellowpages.com/internet/terminology.html). When a number of protocols are used together, they are called a **protocol suite**.

The protocol suite that was developed for the Internet is called **TCP/IP** and its name is a combination of the names of the two most heavily used protocols in the suite (**TCP** stands for **Transmission Control Protocol** and **IP** stands for **Internet Protocol**). The **Internet Protocol** defines a way to uniquely identify computers on the Internet using an addressing system. IP version 4 (**IPv4**), which is currently the most widely used version of the IP protocol, consists of **4 numbers between 0 and 255 separated from each other by a dot**. Examples of IP address include 207.21.94.50, 54.3.59.2, and 204.74.99.100. All the IPv4 addresses from 0.0.0.0 to 255.255.255.255 create an **address space** which contains 4,294,967,296 addresses.

IP version 6 (**IPv6**) is the newest version of the IP protocol and it has an address space which contains 340,282,366,920,938,463,463,374,607,431,768,211,456 addresses! The transition from IPv4 to IPv6 has begun, but it is moving slowly. Most hosts on the Internet will continue to use the IPv4 protocol for a long time and therefore IPv4 is what we will use in this document.

Drawing 13.2 contains the same model of a network that was shown in Drawing 13.1 but with **IPv4** addresses assigned to each computer:



Drawing 13.2: IP Addresses.

If PC #3 needed to send a message to PC #4, the IP address of PC #4 (which is 207.21.94.214) would be placed into the message. The IP address of the sender (207.21.94.72) is also placed into the message in case PC #4 needs to send a reply (this is similar to placing a return address on a letter). PC #3 will then send the message to the switch, the switch will look at the message's destination address and then pass the message to PC #4.

If one of the computers on this local network needs to send a message to a computer which is not on the LAN, then the message is sent to the **gateway** computer and the gateway will then route the message to the Internet.

13.1.3 Clients and servers

On LANs and on the Internet, there are a number of ways for communications between computers to be organized and these organizations are often called **architectures**. One architecture is called **Peer-to-Peer** (P2P) and it treats computers on the network as equals that exchange information with each other. An example of a P2P application is instant messaging.

Another architecture that is used with networked computers is called **Client-Server**. With a Client-Server architecture, a **server** is a computer that accept requests from other computers on the network, performs the work that was requested, and returns the results of the work to the requester. A **client** is a computer that sends a request to a server, receives a response, and then makes use of the information that was contained in the response.

In the LAN shown in Figure 11, there are 3 servers (a DHCP server, a DNS server, and a SAGE server) and 4 clients. The DHCP and DNS servers will be discussed in the next two sections.

13.1.4 DHCP

DHCP stands for **Dynamic Host Configuration Protocol** and its purpose is to allow computers on a LAN to automatically be configured when they are booted up with the information they need to access the network. This information includes an **IP address**, the **address of the gateway**, and the **address of a DNS server**. We have already discussed what an IP address is and what a gateway is. DNS servers will be covered in the next section.

What you might be wondering at this point is how a computer that doesn't have an IP address yet (because it is booting up) is able to use the network to contact the DHCP server to obtain an IP address. This problem is solved by having the booting computer send a DHCP **broadcast** message to the LAN. Broadcast messages are not sent to any specific machine on a LAN. Instead, broadcast messages are sent to the LAN as a whole and all the computers that are on the LAN receive the message.

If a DHCP request message is broadcast to the LAN, the DHCP server will receive the request at the same time that the rest of the computers do. The other computers will read the contents of the message, see that it contains a DHCP request, and then they will ignore it. The DHCP server, however, will read the contents of the message, see that the message was meant for it, and send DHCP configuration information back to the sender.

13.1.5 DNS

Each of the millions of computers on the Internet can be accessed using their IP addresses. For example, the IP address the server that contains the sagemath.org website is **128.208.160.192**. You can access this website directly by launching a web browser and then entering **http://128.208.160.192/sage** in the URL bar.

It is difficult for humans to remember numerous numbers, however, so a **system for associating names with IP address numbers** was created for the Internet. The name of the system is **DNS** and it stands for **Domain Name System**. A name that is associated with one or more IP address is called a **domain name** and a **domain name** that has a given machine's **hostname** at its beginning (and a period at its end) is called a **fully qualified domain name**. Examples of domain names are:

gentoo.org
yahoo.com
sourceforge.net
google.com
sagemath.org
wikipedia.com

Examples of fully qualified domain names are:

kiwi.gentoo.org.
loon.gentoo.org.
wren.gentoo.org.

DNS is implemented as a large database that is distributed across the whole Internet. Domain names need to be registered with a **domain name registry** organization before they will be entered into the DNS system. Examples of domain name registry companies include godaddy.com, networksolutions.com, and register.com.

The DNS server on the LAN in Figure 12 has three functions. The first function is to accept messages that contain **domain names** from clients and to return the **IP address** that are associated with these names. When a user types in a domain name like **sagemath.org** into a browser's URL bar, the browser cannot contact the SAGE website server yet because it does not know its IP address. The operating system that the browser is running on will therefore send the domain name to the DNS server (using the DNS server's IP address that it obtained through DHCP) and the DNS server will respond with one or more IP address that are associated with the sagemath.org domain name. The system will then use one of these IP address to contact the server that the SAGE website is on.

The second function that a local DNS server has is to **define** the **domain name** to **IP address** mappings for the machines on the local network. If a remote computer on the Internet wants to know the IP address for a machine on the local network, and its DNS server does not know the mapping, the remote DNS server will contact the local **authoritative** DNS server to ask what the mapping is. The remote DNS server will then remember this mapping for a certain time in case machines on the remote network need to know the mapping in the future.

The third function that a DNS server has is to take messages that contain **IP addresses** and return the **domain names** that are associated with these addresses.

13.1.6 Processes and ports

Now that we have discussed some of the more important technologies that are related to the Internet, it is time talk about what happens when IP messages (referred to as messages from now on) arrive at a computer and what generates messages before they are sent from a computer.

Almost all modern personal computers can have multiple programs running on them concurrently. Here is a list of programs that may be running concurrently on a typical user's computer:

- Web browser.
- Instant message client.
- Word processor.
- File download utility.
- Audio file player.
- Computer game.

In most computers operating systems running programs are called **processes**. In Windows, a list of all the **processes** that are currently running can be seen by running the Task Manager, which is launched by pressing the <ctrl><alt>and<delete> keys simultaneously. On UNIX-based systems like Linux, a list of the running processes can be obtained by executing a **ps -e** command. Here is the list of process that were running on a Linux system which I ha

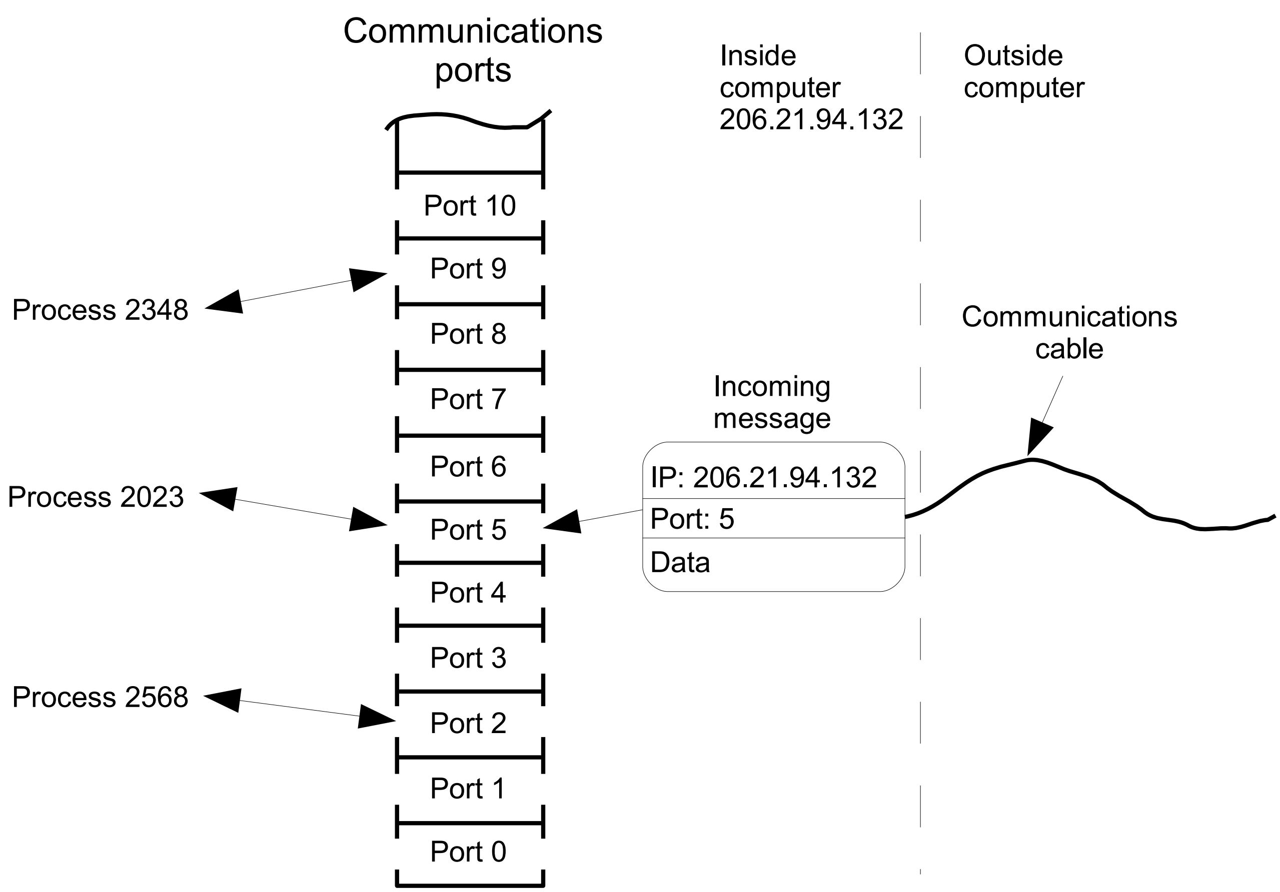
```
manage@sage:~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
    2 ?        00:00:00 ksoftirqd/0
    3 ?        00:00:00 watchdog/0
    4 ?        00:00:00 events/0
    5 ?        00:00:00 khelper
    6 ?        00:00:00 kthread
    8 ?        00:00:00 kblockd/0
    9 ?        00:00:00 kacpid
   10 ?        00:00:00 kacpi_notify
   67 ?        00:00:00 kseriod
  100 ?        00:00:00 pdflush
  101 ?        00:00:00 pdflush
  102 ?        00:00:00 kswapd0
  103 ?        00:00:00 aio/0
 1545 ?        00:00:00 scsi_eh_0
 1547 ?        00:00:00 scsi_eh_1
 1728 ?        00:00:02 kjournald
 1796 ?        00:00:00 logd
 1914 ?        00:00:01 udevd
 2611 ?        00:00:00 shpchpd
 2620 ?        00:00:00 kpsmoused
 3208 tty2     00:00:00 getty
 3209 tty3     00:00:00 getty
 3210 tty4     00:00:00 getty
 3211 tty5     00:00:00 getty
 3212 tty6     00:00:00 getty
 3263 ?        00:00:00 dd
 3265 ?        00:00:00 klogd
 3345 ?        00:00:14 vmware-guestd
 3381 ?        00:00:00 sshd
 3404 ?        00:00:00 atd
 3414 ?        00:00:00 cron
 3959 ?        00:00:00 dhclient3
 4140 tty1     00:00:00 login
 4141 tty1     00:00:00 bash
 4429 ?        00:00:00 syslogd
 4507 ?        00:00:00 sshd
 4508 pts/1    00:00:00 bash
 4538 tty1     00:00:00 sage
 4541 tty1     00:00:00 sage-sage
 4554 tty1     00:00:00 python
 4555 tty1     00:00:05 sage-ipython
 4573 tty1     00:00:00 sh
 4574 tty1     00:00:00 sage
```

```
 4580 tty1     00:00:00 sage-sage
 4591 tty1     00:00:02 python
 4592 pts/2    00:00:00 sage
 4600 pts/2    00:00:00 sage-sage
 4611 pts/2    00:00:06 python
 4611 pts/1    00:00:00 ps
```

If you look towards the bottom of this list you can see SAGE running along with the SAGE Notebook server. Notice that the **ps** command included itself in the list because it was running at the moment that the list was created.

There are four columns in this listing. Each process is given a unique **Process ID** (PID) number when the process is created and these numbers are listed in the **PID** column. The **TTY** column indicates whether or not a process is attached to a terminal and if it is, what terminal it is attached to. The **TIME** column indicates how much CPU time the process has used so far in hours, minutes and seconds .

When a message arrives at a computer from the network, the computer must decide which process to give the message to. The way that the TCP/IP protocol solves this problem is with software-based communications **ports**.

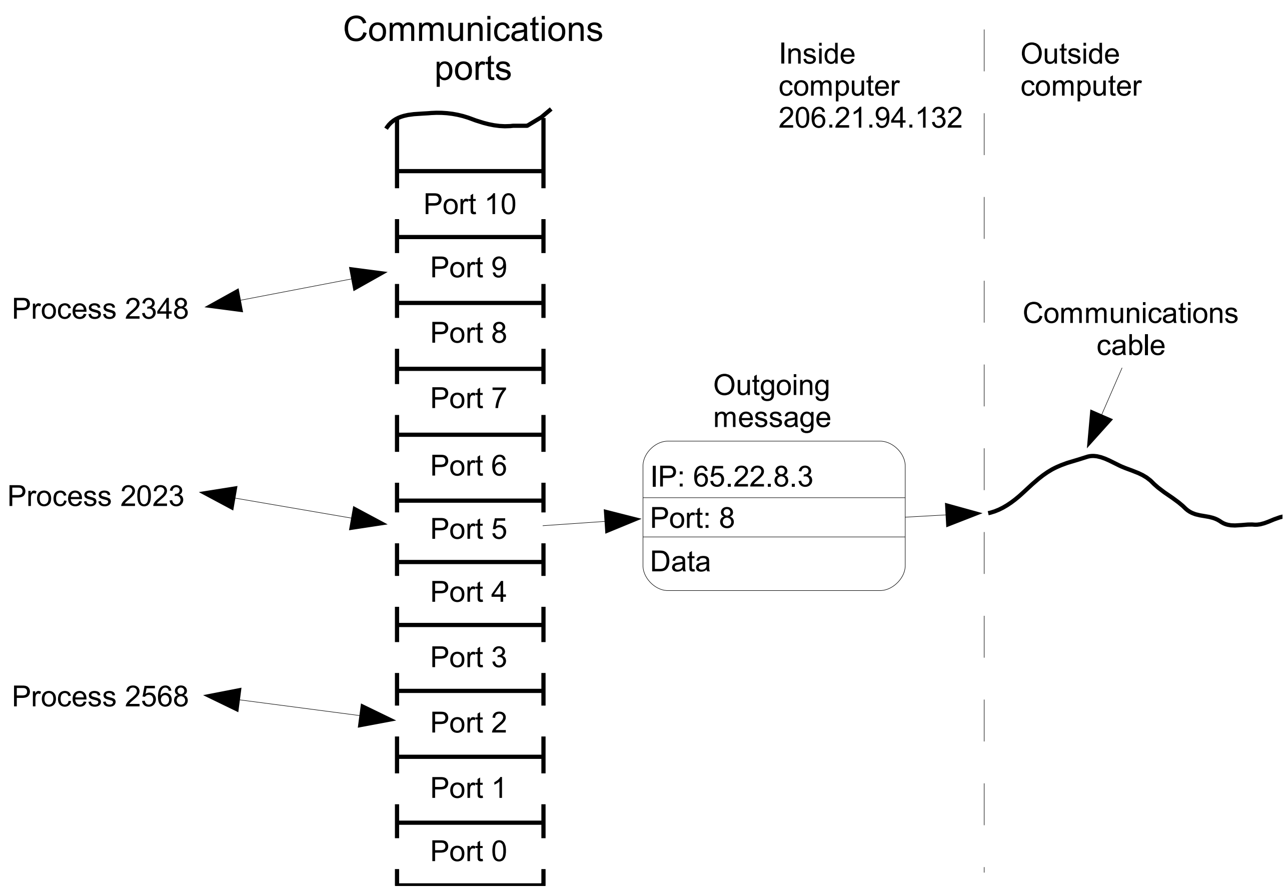


Drawing 13.3: Communications ports.

Drawing 13.3 shows the inside and the outside of a computer that is connected to a network and which has an IP address of **206.21.94.132**. The communications ports are placed between the processes that are running on the left and the network connection on the right. Each port is given a unique number with the lowest port number being **0** and the highest port number being **65535**. Each message that arrives from the network has a port number included in it so that the system knows which port to send the message to.

In Drawing 13.3, a message which has **port 5** as its destination port has arrived from the network and therefore the system will place this message into **port 5**. **Process 2023** has been bound to **port 5** and, when the system sends the message to this port, **process 2023** will take the message and then do something with the information it contains.

Drawing 13.4 shows a message from process **2023** being sent to another computer on the network which has an IP address of **65.22.8.3**. When this messages arrives at the destination computer, it will place the message into it's **port 8** and hopefully there is a process at that computer which is bound to **port 8**.



Drawing 13.4: An outgoing message.

13.1.7 Well known ports, registered ports, and dynamic ports

Now that you know what ports are and how processes are bound to them, you may be wondering how people determine which processes should be bound to which ports. An organization called **IANA** (Internet Assigned Numbers Authority) is responsible for various number schemes associated with the Internet and one of them is the TCP/IP port scheme. IANA has divided the **0 - 65535** port range into the following three address blocks:

0 - 1023 -> Well Known Ports.

1024 - 49151 -> Registered Ports.

49152 - 65535 -> Dynamic and or Private Ports.

13.1.7.1 Well known ports (0 - 1023)

A list is maintained by IANA which indicates which kinds of programs are usually bound to specific port numbers in this range. For example, **web servers** are bound to **port 80**, **SSH (secure shell) servers** are bound to **port 22**, **FTP (File Transfer Protocol servers** are bound to **port 20**, and **DNS** servers are bound to port **53**. Here is a list of the first 25 well known ports and the full list can be obtained at http://www.iana.org/assignments/port-numbers:

Keyword	Decimal	Description	References
-------	-------	-----------	----------
	0/tcp	Reserved	
	0/udp	Reserved	
#		Jon Postel <postel@isi.edu>	
tcpmux	1/tcp	TCP Port Service Multiplexer	
tcpmux	1/udp	TCP Port Service Multiplexer	
#		Mark Lottor <MKL@nisc.sri.com>	
compressnet	2/tcp	Management Utility	
compressnet	2/udp	Management Utility	
compressnet	3/tcp	Compression Process	
compressnet	3/udp	Compression Process	
#		Bernie Volz <volz@cisco.com>	
#	4/tcp	Unassigned	
#	4/udp	Unassigned	
rje	5/tcp	Remote Job Entry	
rje	5/udp	Remote Job Entry	
#		Jon Postel <postel@isi.edu>	
#	6/tcp	Unassigned	
#	6/udp	Unassigned	

```
echo              7/tcp    Echo
echo              7/udp    Echo
#                          Jon Postel <postel@isi.edu>
#                 8/tcp    Unassigned
#                 8/udp    Unassigned
discard           9/tcp    Discard
discard           9/udp    Discard
#                          Jon Postel <postel@isi.edu>
discard           9/dccp   Discard SC:DISC
#                          IETF dccp WG, Eddie Kohler
<kohler@cs.ucla.edu>, [RFC4340]
#                10/tcp    Unassigned
#                10/udp    Unassigned
systat           11/tcp    Active Users
systat           11/udp    Active Users
#                          Jon Postel <postel@isi.edu>
#                12/tcp    Unassigned
#                12/udp    Unassigned
daytime          13/tcp    Daytime (RFC 867)
daytime          13/udp    Daytime (RFC 867)
#                          Jon Postel <postel@isi.edu>
#                14/tcp    Unassigned
#                14/udp    Unassigned
#                15/tcp    Unassigned [was netstat]
#                15/udp    Unassigned
#                16/tcp    Unassigned
#                16/udp    Unassigned
qotd             17/tcp    Quote of the Day
qotd             17/udp    Quote of the Day
#                          Jon Postel <postel@isi.edu>
msp              18/tcp    Message Send Protocol
msp              18/udp    Message Send Protocol
#                          Rina Nethaniel <---none--->
chargen          19/tcp    Character Generator
chargen          19/udp    Character Generator
ftp-data         20/tcp    File Transfer [Default Data]
ftp-data         20/udp    File Transfer [Default Data]
ftp              21/tcp    File Transfer [Control]
ftp              21/udp    File Transfer [Control]
#                          Jon Postel <postel@isi.edu>
ssh              22/tcp    SSH Remote Login Protocol
ssh              22/udp    SSH Remote Login Protocol
#                          Tatu Ylonen <ylo@cs.hut.fi>
telnet           23/tcp    Telnet
telnet           23/udp    Telnet
#                          Jon Postel <postel@isi.edu>
                 24/tcp    any private mail system
```

```
                      24/udp    any private mail system
#                               Rick Adams <rick@UUNET.UU.NET>
smtp                  25/tcp    Simple Mail Transfer
smtp                  25/udp    Simple Mail Transfer
```

When one computer on the network wants to make use of a specific service that is running on another computer on the network, the first computer creates a message, places the port number of the desired service into the message, and then sends it to the destination computer. If a process that implements the well known service for that port is bound to the port, then this process will receive the message and perform the requested work.

The main restriction on **processes** that are bound to ports in the well known ports range is that they **must** be running with **super user privileges**.

13.1.7.2 Registered ports (1024 - 49151)

Registered ports work similarly to **well known ports** except that the **processes** that are bound to them **do not** need to be running with **super user privileges**. The list of **registered ports** is included in the same **IANA document** that contains the list of **well known ports**.

13.1.7.3 Dynamic/private ports (49152 - 65535)

These ports are used as needed and they do not have any specific type of process associated with them. A typical use of the ports in this range is for a web browser to make an outgoing connection with a web server.

13.1.8 The SSH (Secure SHell) service

An example of a service that makes itself available through a well known port is the **SSH** (Secure SHell) service and it is usually bound to port **22**. The **SSH service** allows a person to log into one computer on a network from another computer on the network. The person must know the **username** and **password** for an account on the remote machine before logging into it and the remote machine must have a SSH service (in the form of a process) running and bound to port 22. SSH is able to provide a secure connection between the machines by encrypting the data that is passed between them.

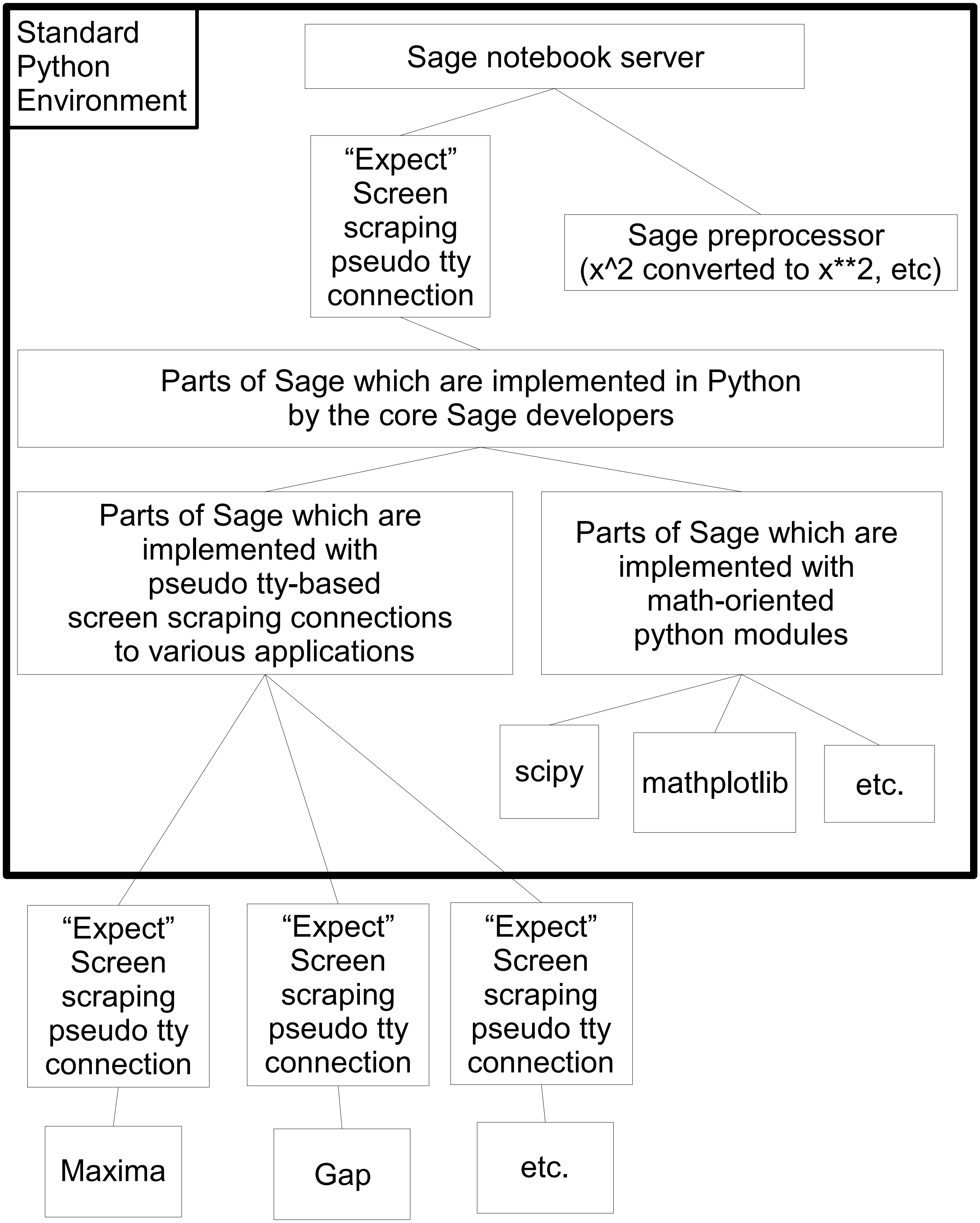
On UNIX-based systems, the SSH client program is simply called SSH and on Windows systems you can download and install a program called **putty.exe** that will allow you to remotely log into a machine that is running the ssh service. The **putty.exe** program can be downloaded from (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html).

When the **ssh** client program is asked to log into a remote machine for the first time, it tells the user that it does not currently have encryption information for this host and asks if it should continue. Answer by typing the word "**yes**". The program then indicates that it added information about this host to a known hosts list and it will not ask the question again in the future.

13.1.9 Using scp to copy files between machines on the network

The SSH service is not only able to allow a user to log into a remote machine, it can also be used to copy files between machines on the network. The Linux client program for copying files is called **scp** (Secure Copy) and a popular **Windows scp** client, called **pscp.exe**, can be obtained from the same url that **putty.exe** was.

13.2 SAGE's Architecture (in development)



Drawing 13.5: SAGE's architecture.

13.3 Linux-Based SAGE Distributions

(In development...)

13.4 The VMware Virtual Machine Distribution Of SAGE (Mostly For Windows Users)

(In development...)