An Invitation to Programming

A "textbooklet" to accompany INF 120
Northern Kentucky University

Version: January 1, 2011
Minor updates: February 28, 2015

Kevin Kirby kirby@nku.edu

Contents

0 Preface

Science is about acquiring and organizing knowledge in order to understand, predict, and manage the world. **Computer science** focuses on the "organizing" part of this: the world is complex, and we need a science that can help us make the best use of computation as we confront this complexity.

Computer science has been around for over two millennia. Most of that time, however, computation was limited to what humans could do by (for example) moving beads on wires or making marks on paper. The term "computer" was a job description, just like "carpenter" or "plumber": a computer was a person who did lots of arithmetic to help scientists or engineers with their work. Only in the middle of the 20th century, with the creation of digital computing machines, did this science start to mature.

One way to begin understanding computer science is to learn to write programs. **Programs are pieces of text that tell digital devices what to do.** Programs are what make smart devices smart. Earlier generations used the word "computer programming" for this, since what people wrote programs for were computers, but this has become too narrow. Your refrigerator, your phone, and your car can all be smart devices nowadays, but it is not very helpful to call them "computers".

We refer to the text of programs as **code**. Today's digital world runs on code. **Your goal in this course is to learn to build simple things in code.** Learning to write code is fascinating and empowering, but it is not especially easy. Therefore, this course is meant as a gentle introduction to programming, an invitation to computer science. It is meant for **all** students, not just for majors in computing.

This document is not quite a textbook, but it is more than just notes. Let's call it a **textbooklet**. It is short and to the point because you are meant to read **all** of it, not just skim through it. **It does not stand alone**. You will be using it together with a set of highly visual/pictorial slides, as well as code examples.

If you would like to have a full textbook on this material, you should use this book:

> *Introduction to Computing and Programming in Python: A Multimedia Approach*, 4th Edition
> by Mark Guzdial and Barbara Ericson. Pearson Prentice Hall, 2015.

This course is designed for students who are comfortable with mathematics at the level of ninth grade algebra, and who have what is now a junior-high level of computer literacy. Specifically, it assumes that you are comfortable with the basic use of Windows or the Mac OS (in particular you understand files and folders), know your way around a web browser, and have edited documents in a word processor.

The course and this textbooklet both require that you use the **Jython Environment for Students** ("JES"), a piece of free software that will help you write your own programs. Our examples are based on JES version 5.0. It is available at https://github.com/gatech-csl/jes/releases.

This textbooklet is a work in progress. It originated in Spring 2007 when NKU first offered its introductory course using JES. It stabilized in 2010 when it was first offered as a general education course at NKU, with minor revisions in 2011 and very minor updates in 2015. If you see typos or have comments or suggestions for improvement, we would be very pleased if you contact your teacher or kirby@nku.edu.

1 Introduction

A. How does code work?

Code is written in **a programming language**. Human languages like English or Chinese are very complex, and meanings of sentences can be fuzzy. By contrast, programming languages are deliberately designed to be simple, and to allow programmers to say what they mean with perfect clarity. This is, as you may suspect, not very natural.

By one account[1], the five most popular programming languages as of 2014 are Java, C, C++, C# ("C sharp"), and Python.

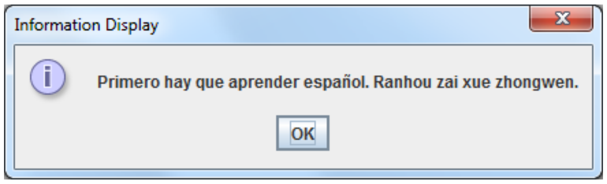
Our goal here is to introduce programming using the **Python** language. Aside from its popularity in industry, we use it here because it makes it easy to do contemporary programming examples involving digital media. For example, many of our examples, particularly in the second half of the course, will involve manipulating digital photos and digital audio[2].

A program consists of **statements** in a programming language. Programs can be very large, consisting of millions of statements, written by teams of people over many years. On the other hand, there are also useful programs that are very small, maybe only a dozen statements. Very small programs are often called **scripts**. Some programming languages, such as C++ and Java, are designed to be used to write very large programs. Other programming languages, such as Python and Ruby, are typically used to write scripts.

In most programming languages, statements are **commands**. Here is a line of Python:

```
showInformation( "Primero hay que aprender español. Ranhou zai xue zhongwen." )
```

This is a command to pop up a little box containing the message given in quotes. The result of this command would look like this:



```
Primero hay que aprender español. Ranhou zai xue zhongwen.
```

You might ask: if this is a command, exactly who or what is supposed to obey the command? You could answer, "the computer." In reality, it is more subtle. Let us take a closer look.

Suppose you are running this on a notebook computer, say a Dell Inspiron 15R, to be concrete. Inside this notebook you have a **processor** made by Intel Corporation, a so-called "Core I3". This processor contains hardware to do all your computing (two Central Processing Units, or **CPU**s) as well as hardware to do graphics (a Graphics Processing Unit, or GPU). If you have Windows installed on your notebook computer,

[1] Source: http://spectrum.ieee.org/computing/software/top-10-programming-languages. The top five *human* languages are, incidentally: Mandarin Chinese, Spanish, English, Hindi/Urdu, and Arabic.

[2] To learn about this approach, you can read this document written for the National Center for Women in Information Technology: http://www.ncwit.org/sites/default/files/resources/mediacomputationgeorgia_tech_attractingstudentsengagingintroductorycomputingcurriculum.pdf.

then Windows is controlling this hardware. Windows is an **operating system**, which is software which manages the hardware and runs programs.

When Windows runs your programs, it expects them to be compatible with the hardware. This means they should be expressed as instructions in a language that the processor understands. In the case of the most popular processors at present, including the Core I3, this language is called the "x86 instruction set." Unfortunately, reading and writing x86 code is practically impossible for humans; it consists of very long sequences of 0s and 1s. Fortunately, software comes to the rescue: there is software that will allow us to write our programs in the language of our choice (say C++ or Python) and have it translated into x86 code for Windows to run. Such programs are called **compilers** (if they translate a whole program at once) or **interpreters** (if, like human interpreters at the United Nations, they translate the language statement-by-statement right when it is needed).

So, I can write a computer game, call it MeinKraft, in the C++ language. I can then have a compiler translate it to x86 code, and then I can sell that code to consumers. As they play my game, they are running the x86 version of my program (which I could never make sense of directly) on their processors.

So, similarly, if I write that one-line Python program above, I can translate it into x86 code and have Windows run it on the computer directly. But…. there is another way.

Instead of having Windows take charge of running my programs, I can have Windows *run a program whose job it is to run programs*. One such program that runs programs is called the **Java Virtual Machine**. The Java Virtual Machine runs programs in its own special language, known as "Java bytecode". Java bytecode is meant to be an improvement on x86 code. The Java Virtual Machine itself is written in x86 code, and runs on Windows. Do you see the layers here? This is a deep idea, and students who major in computer science end up learning a lot about building things in layers.

So now programmers have another option. Instead of writing a program that will be translated into x86 code for Windows to run, they can write a program that will be translated into Java bytecode for the Java Virtual Machine to run. Programmers who write in the Java language do this. And, in this course, we will do this in the Python language. **Jython** is the "dialect" of Python designed for programs that are meant to run on the Java Virtual machine. The program **JES**, the Jython Environment for Students, allows us to edit programs, have them translated into Java byte code, and then have the Java Virtual Machine run them.

So, in summary:

- You write a program in the Jython dialect of Python, using JES.
- JES gets the Java Virtual Machine to run it.
- Windows runs the Java Virtual Machine on your Intel processor.

☐ Practice A

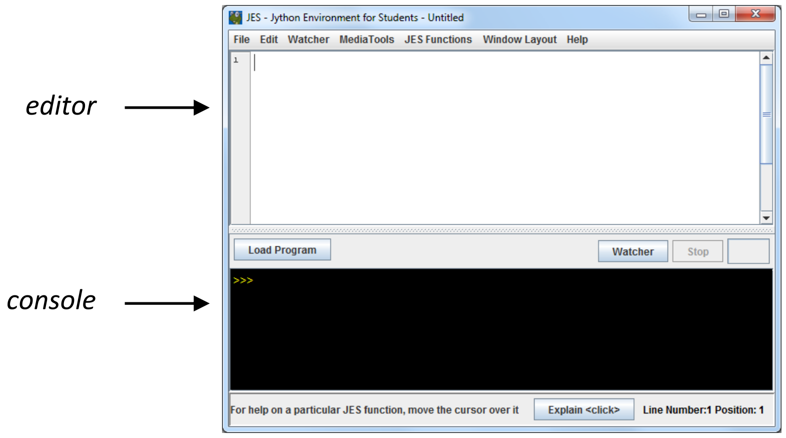
A1. Type "python programming" into your favorite search engine and surf some of what shows up.

B. Writing code

In your first lab session you will practice using JES. There are two different ways to run your Python code.

Directly at the console: You can type Python code into the so-called "Interaction Area" at the bottom of the JES window. We will call it the **console** for short, a more common term.

Indirectly in the editor: You can type Python code into the so-called "Program Area" at the top of the JES window. Like a word processor, you can, create, open and save files here. We will call it the **editor**.
Except for running single-line commands, we will exclusively use the editor instead of the console.


editor
console
JES - Jython Environment for Students - Untitled
File Edit Watcher MediaTools JES Functions Window Layout Help
Load Program
Watcher
Stop
>>>
For help on a particular JES function, move the cursor over it
Explain <click>
Line Number:1 Position: 1

When you work on a Python program in the JES editor, you should save it to a file that ends in the suffix **.py**. If you are using Windows, you should make sure you set up your file browser to make these suffixes visible[3]. You will find that JES automatically creates other files ending in **.pybak** and **.pylog**. You do not need to retain these files. When you are asked to submit a program on an assignment, you will submit just the **.py** file.

Most of the programs in the first half of the course will be done this way.

Let's work on a little project: a program that pops up a box that says "Monty Python".

1. Launch JES.

2. Type the following code into the editor window. The second line is indented by two spaces.

```
def main():
  showInformation( "Monty Python" )
```

3. Save it to a file. Let's call it **MyFirstProgram.py**.

4. Hit the **Load** button.

JES says "Loading Program" on the console, but nothing else seems to have happened. So let's pause here. Your program consists of one **definition**. What is being defined here is something called a **function**, a little packet of code. The name of this function is called **main**. It consists of only one line of code, the command to show the little information box.

If you actually want to run this command (we often say "**execute** this command"), you have to command JES to run this function called main. Here's how:

[3] In Windows 7, when browsing Documents pull down *Organize* (the leftmost menu), select *Folder Options*, choose the *View* tab, and un-check the box "☐ Hide extensions for known file types."

5. At the console prompt (“>>>”) type:

```
main()
```

Now your **main** function executes, and you will see the information box.

As you write programs, you will almost certainly make mistakes. There are several different kinds of mistakes you can make. A mistake in a program is called a **bug**. Finding and correcting mistakes in programs is called **debugging**.

First, you could do something that gets the grammar of the Python language wrong:

```
def main:
  showInformation( "Monty Python" )
```

JES will find this mistake when you hit the **Load Program** button, highlighting it in yellow and giving you an error message on the console. The error message is often not very specific, and you will need to develop some experience in spotting your errors. Do you see what was wrong here? You left out the parentheses **()** after **main**. This kind of error is called a **syntax error**. *Syntax* refers to the arrangement of symbols in a program, from an Ancient Greek word used to refer to the ordering of troops in formation on a battlefield.

Second, you could do something that is grammatically correct but is meaningless:

```
def main():
  showInformashun( "Monty Python" )
```

When you hit the Load Program button JES does not complain. However, when you run the program (by typing **main()** at the console as usual) you get an error: it says the name “showInformashun” is not found. This is true. JES provides a function named *showInformation*; it could have provided one called *showInformashun* if someone had written it, but no one did. So the line is meaningless.

Third, you could do something that is grammatically correct, meaningful, but does not meet the requirements:

```
def main():
  showInformation( "Monty Hall" )
```

This program runs just fine, but it is not what was asked for, so in that sense, it is still a bug.

Finally, not all variations of this code are errors. For example, you can introduce or remove “white space” in certain cases without any problem:

```
def main( ):
  showInformation ("Monty Python")
```

Part of learning any language is learning its syntax. This will be a significant part of this course, though, you will see, it is not the hardest part.

☐ Practice B

B1. Introduce each of the following changes into the program in this section (separately). Are they bugs? If so, does JES spot them?

a. Don't indent the second line.
b. Leave out the closing quotation marks.
c. Change **showInformation** to **ShowInformation**
d. Leave out the closing parenthesis.
e. Put in two colons :: on the first line.
f. Change **def** to **define** on the first line.
g. Change main to florida, but run the program by typing **florida()** at the console.

B2. Create and run the following program:

```
def main():
  showInformation( "Monty Python" )
  showInformation( "Python moan" )
```

Do the two information boxes appear on your desktop together?

B3. Turn on your speakers and make sure the volume is not zero. Create and run the following program:

```
def main():
  playNote( 50, 1000, 127 )
  playNote( 51, 1000, 127 )
  playNote( 52, 1000, 127 )
```

Do you hear anything? (If not, make a note of this and tell your instructor.)

C. Algorithms

We write programs to do something. There are often many ways to do something right, and many ways to do something wrong. This means we have to have a clear step-by-step description of how to accomplish what we want to do. Such a clear step-by-step description is called an **algorithm**.

An algorithm is like a recipe. You can use a human language like English to give a recipe, as long as you use it precisely. For example:

Cook up some onions

is probably not precise enough to be considered a recipe. But, still using ordinary English, we can be more precise:

Dice one onion and cook with 1 teaspoon of canola oil in a frying pan until slightly brown.

Recipes usually deal with food (though not necessarily), and algorithms usually deal with computation (though not necessarily). Here is an algorithm for translating a word into Pig Latin:

```
If the word starts with a vowel
        put a 'w' on the end of the word,
otherwise
        move the first letter to the end of the word.
Now add "ay" to the end of the word.
```

For example, "dog" becomes "ogday", and "elk" becomes "elkway".

Here is an algorithm for finding a matching pair of socks in a basket of laundry:

```
Reach into to basket and grab a sock
Repeat:
        Grab another sock.
        If it doesn't match the first one you grabbed
                put it back in the basket
        otherwise
                you are done.
```

These examples illustrate some key points about algorithms:

1. They involve following instructions **in order**.
2. They often involve checking **conditions** to make **decisions**.
3. They often involve **repetition**.
4. They are often poorly designed. Can you spot some flaws in the sock-finding algorithm?

Programming languages allow the precise expression of algorithms so that computers can **execute** them. That means all programming languages allow for doing something in order ("sequencing") as well as allowing for decisions and repetition. (In fact, with those three features plus some memory you can do just about *anything*.) But programming languages also won't keep you from writing bad algorithms.

In this course, you learn to write code. But code is meaningful only insofar as it follows an algorithm, so a lot of learning to program is learning to create and reason about algorithms.

☐ Practice C

C1. Write a better version of the sock-finding algorithm.

Learning Outcomes for Chapter 1

Reminder: As we pointed out in the preface, reading this textbooklet by itself is not meant to be sufficient for achieving all of these learning outcomes. You should attend class, review the slides, and above all *practice practice practice*.

☐ Be able to explain the following terms:

- programming language
- program
- script
- operating system
- central processing unit (CPU)
- compiler
- interpreter
- Java Virtual Machine
- Jython
- bug
- debugging
- syntax
- algorithm

☐ If someone gives you the text of a short Python program, be able to use JES to enter it and run it.

2 Types and Variables

A. Data types

Programs operate on **data**. Data is how information is packaged. Casually speaking, data can be names, social security numbers, photos, music, and so on.

At the lowest level of computer hardware, there is only one kind of data: bits. A **bit** is something that can take on only two values. We usually use 0 and 1 to represent these two values. Ultimately, everything in the digital world (from phone numbers to movies) is represented as bits. Every piece of digital data is something that looks like 0101011101010111111010101011111100000101110100 1.

But if we want to write programs to manipulate data, operating on 0s and 1s is extremely inconvenient. Programming languages allow us to manipulate data described in more useful forms. In other words, they allow our programs to deal with other types of data besides bits.

This course will give you more practice with different types of data than in traditional introductory programming courses. Our programs will deal with the following data types:

Integers (whole numbers like 45)
Strings (like "hello there")
Colors
Digital images
Floats (real numbers like 3.1415)
Pixels (the colored dots in digital images)
Digital sounds
Sound samples

In the next three sections we will introduce important ideas (literals, variables, and operations) on the two most important data types, integers and strings. Then we will provide very basic introductions to some of the other types mentioned here.

B. Literals

Here is a very simple Python program:

```
def main():
  printNow( 45 )
  printNow( "hello there!" )
```

It prints out two pieces of data on the console. Each of these two pieces of data is of a different type. The first one is an integer, the second is a string.

In this program, these two pieces of data have specific values which we have typed in to the code, shown in red. These are called **literals**. For example, **4** is known as an **integer literal**. And "hello there" is known as a **string literal**. (Beware: "1974" is a string literal, which is very different from 1974, the integer literal!)

Type in this code, save it, load it, and run it. How does what you see printed on the console differ from the literals you typed into your code?

In addition to literals, we can refer to data using *variables*, described in the next section.

DEEPER

Integer literals cannot have commas. Instead of writing **1,405,972** you must write **1405972** with no commas or decimal points.

String literals must be enclosed in double quotes, as in "Hey!" In Python, you may also use single quotes (like 'Hey!'), but using double quotes is a better habit, since you will probably go on to study languages like Java, C++ or C#, where double quotes are required.

C. Variables

Do you remember variables from algebra? They were single letters (like **x**,**y** or **z**) standing for numbers. For example, you learned to solve equations like 2**x** + 4 = 10, and found that **x** = 3.

In programming languages, variables are more flexible than what you learned in algebra, in two ways.

- You can use names like **weight**, **energy**, **salary**, instead of just single letters like **x**,**y** or **z**.

- They can stand for things besides just numbers. In algebra you could have a variable **y** that had the value 51. You can have that in a program too. But you can also have a variable **firstName** that takes on the value "Ursula".

Type, save, load and run the following Python program:

```
def main():
  age= 5
  printNow( age )
  age= 16
  printNow( age )
  age= 76
  printNow( age )
```

Here you can see one variable, **age**, that takes on three different literal values.

Can you see the meaning of the lines with equals signs? These are called **assignment statements**. They are commands to **change** the value of a variable. That is, the line

```
age = 5
```

means

make the variable ***age*** *equal to the value* ***5***

Now type, save, load and run the following little Python program:

```
def main():
        printNow( age )
```

When you run it, you will see an error message. Why? Because you are asking it to print the value of the variable **age**. But **age** has never been given a value! It is said to be "undefined."

DEEPER

Names for variables must start with a letter, and can contain letters, digits, and underscores. Here are some variable names: **amount**, **photo128b** , **max_annual_salary**, and **maxAnnualSalary**. Some names are reserved and you can't use them for variables. Examples include **raise**, **elif**, and **while**. There are a couple dozen of these special "reserved words"; you'll learn them later.

Variable names are case sensitive, meaning uppercase and lowercase are regarded as different. So **amount** and **Amount** and **aMoUnT** are three different variables with absolutely no relationship to each other.

Make sure you don't confuse the name of variables with strings that are spelled the same way:

```
age= 22
printNow( age )
printNow( "age" )
```

The output of this program is:

```
22
age
```

☐ Practice C

C1. There are bugs in the following Python programs. Explain what precisely what the errors are.

```
def main():
  age= 10
  printNow( Age )
```

```
def main():
  age= 24,124
  printNow( age )
```

```
def main()
  age= 10
  printNow( age )
```

```
def main():
  annual Salary = 48000
  printNow( annual Salary )
```

D. Operations on data

When we think of different types of data, we realize we can transform that data in different ways. These are called **operations**.

> *Some things we do to integers (whole numbers):*
> Add them, subtract them, multiply them, etc.
>
> *Some things we do to strings:*
> Join them together, repeat them, etc.

With integers, we use **+** for addition, **–** for subtraction, ***** for multiplication, and **/** for division. For example, 4*3 is 12. Enter and run the following example program to see that this works as expected.

```
def main():
  printNow( 100 + 5 )
  printNow( 100 - 5 )
  printNow( 100 * 5 )
  printNow( 100 / 5 )
  printNow( 2*(10/5)+7) )
```

With strings, we use the + sign for the operation of joining together. For example “hot” + “cakes” is “hotcakes”. Enter and run the following example program to see that this works as expected.

```
def main():
  printNow( "de" + "li" + "cious" )
  printNow( "15" + "21" )
```

Do you see why the output in the last line is 1521, not 36? This is very important. It is important to keep data types straight. You don’t want to confuse integers and strings, and you certainly cannot add them together.

> 7 + 8 is 15
> “7” + “8” is “78”
> “7” + 8 is an error!

☐ Practice D

D1. Study the following program (don’t run it yet!).

```
def main():
  a= 0
  b= 5 * 10
  printNow( a + b )
  printNow( b + (2*a) )
  printNow( a – (100/(b-30)) )
```

What is the output ? If we change the first line to **a=100**, what is the output ? Check your answers by typing in, saving, loading and running this program.

D2. Study the following program (don't run it yet!) and say what the output will be.

```
def main():
  sA= "hi"
  sB= "ho"
  printNow( sA + sB )
  printNow( sB + sA + "ha" )
  printNow( "10" + "5")
  printNow( "sA" + sA )
  printNow( "+" + "+" )
```

Check your answers by typing in, saving, loading and running this program.

E. Floats

In many applications we need numbers that are not integers, such as 3.141 or 0.000074. Mathematicians would call these real numbers. Computer scientists call them floating point numbers, or just **floats** for short.

The literal 4.0 is a float literal; is not the same as 4, which is an integer literal. Yes, they are *mathematically* equivalent, but we will see later that it is useful to maintain this distinction.

F. Pictures

Digital images are an important data type nowadays. In JES, this data type is called **picture**. Pictures consist of colored dots called **pixels**. Here is a Python program that creates an empty white square picture (500 pixels wide and 400 pixels tall) and shows it.

```
def main():
  pic= makeEmptyPicture( 500,400 )
  show( pic )
```

Here we are using a variable named pic to hold the picture. Run the program to see how it works.

G. Colors

Color is an important data type associated with pictures. The human eye can distinguish millions of colors. In JES programs there are literals for the thirteen most common colors: b**lack,blue,cyan,darkGray,gray, green,lightGray,magenta,orange,pink,red,white,yellow**.

The following program is just like the program above, except that the picture is pink:

```
def main():
  pic= makeEmptyPicture( 500,400, pink )
  show( pic )
```

Of course, 13 colors is not enough. We need to be able to construct our own colors.

In the digital world, colors are often represented by the relative amounts of red, green, and blue light that need to be combined to make the color. The minimum amount of red, green, or blue a color can have is, of course, 0. The *maximum* amount of red, green, or blue a color can have is **255**. Yes, 255 is a strange number, but it appears frequently in the digital world. (Can you guess why color values run from 0 to 255, instead of something like, say, 0 to 100, which would seem to make more sense?)

Here are the "**RGB**" levels of some different colors. See if this table makes sense to you.

	Red level	Green level	Blue level
red	255	0	0
green	0	255	0
blue	0	0	255
black	0	0	0
white	255	255	255
gray	128	128	128
yellow	255	255	0

How many different possible colors are there, if we define them this way?

The following line of Python creates a color that has a red level of 95, a green level of 150, and a blue level of 190, and stores it in a variable called myFavoriteColor:

```
myFavoriteColor= makeColor( 95,150,190 )
```

We can use this color variable wherever you would have used a color literal. Here is another version of the program above.

```
def main():
  myFavoriteColor= makeColor( 95,150,190 )
  pic= makeEmptyPicture( 500,400, myFavoriteColor )
  show( pic )
```

H. Getting data from the user

Keep in mind the distinction between the **programmer** and the **user**. The programmer is the person who writes the program, the user is the person who runs it. Even when they are the same person, they are two different roles.

Programs often request data from the user. This gives them flexibility. Enter and run the following program[4]:

```
def main():
  name= requestString( "What is your name?" )
  speed= requestInteger( "What is the airspeed of an unladen sparrow?" )
  color= pickAColor()
  printNow( name )
  printNow( speed )
  printNow( color )
```

[4] See http://www.youtube.com/watch?v=y2R3FvS4xr4. Python was named after Monty Python, not the snake.

The first two lines are assignment statements; note the "=" sign.   For example, the first line makes the variable **name** equal to whatever the user entered (through a dialog box with the prompt "What is your name"?)   What output do you see when you print a color at the console?

I. Comments

In addition to being bug-free, a good program should be easy for humans to understand. Programming languages allow programmers to insert **comments** into their code that are meant for human readers, but are ignored when the program runs.

In Python, comments start with the **#** sign. They can be on a line alone:

```
# Using the momentum formula from classical physics, assuming speed is small.
momentum = mass * speed
```

or they can be at the end of a line:

```
momentum= mass * speed     # assumes speed is small
```

In most programming editors, they show up in a different color (JES uses green), to make a visual distinction between comments and real code.

When used judiciously, as above, comments can help explain a program to the human reader. But they are no substitute for making the code itself clear. For example, instead of writing this terrible piece of code:

```
# b is total books, s is books per shelf, x is number of shelves.
tb= s * x
```

you should just write this:

```
totalBooks= booksPerShelf * numberOfShelves
```

Never use comments to simply repeat the code:

```
area= height * width  # the area is the product of the height and the width
```

It is a good idea to put comments at the top of a file explaining what the program will do. All of your assignments will require this.

☐ Practice H

H1. Suppose someone hands you a program that is not working and you see the following line:

```
amount= 12  # set amount to 13
```

What can you conclude? (Think carefully.)

J. Pseudocode

As you write programs, it is often useful to outline them informally first. For example, suppose you were asked to do this:

> *Write a program that asks the user for the width and height of a picture. It then asks the user to pick their favorite color. It then pops up a picture of that size in that color.*

You may wish to outline it this way

```
get the width from the user
get the height from the user
get the color from the user
make a picture with that width, height, and color
show that picture
```

This is not code, since it is not written in a programming language. We call it **pseudocode**, which is just English being used to informally outline an algorithm.

Instead of scribbling this on paper, you may want to type this into JES as comments, making this a starting point for writing your code:

```
def main():
  # get the width from the user
  # get the height from the user
  # get the color from the user
  # make a picture with that width, height, and color
  # show that picture
```

Save this to a file, say **PicMaker.py**. You can then edit this, turning each line into real Python, with some introductory comments at the top of the file.

```
# PicMaker.py
# Creates a solid color picture to the user's specifications.

def main():
  width= requestInteger( "Enter the width" )
  height= requestInteger( "Enter the height" )
  color= pickAColor()
  pic= makeEmptyPicture( width, height, color )
  show( pic )
```

☐ Practice I

I1. Write a program that will create a square colored picture. The program requests two pieces of data from the user: it first asks the user for the color, then it asks the user for the number of pixels on a side of the image (which will be both the width and the height). It then shows the resulting picture.

Learning Outcomes for Chapter 2

☐ Be able to explain the following terms:

- data type
- literal
- variable
- comment
- pseudocode

☐ List four data types and give examples of how they are used.

☐ Be able to write short programs that ask the user for integers, do simple arithmetic on them, and print the results to the console.

☐ Be able to write short programs that request integer, string, and color data from the user, and display them in ways appropriate to each data type.

3 Using Functions

A. Introducing functions

If you look up a recipe for apple pie, it might run like this:

recipe Apple pie:

> *Peel, core and slice 8 apples.*
> *Mix together 1/3 c sugar, ¼ c flour, 1 t cinnamon in a bowl.*
> *Stir in apples.*
> *Combine the 2½ c flour, 2T sugar and ¼ t salt in another bowl.*
> *Add 1/2 c butter and 5t shortening, cutting into small pieces with pastry cutter.*
> *Mix in 8t ice water.*
> *Turn the dough onto a lightly floured surface, knead it together.*
> *Line bottom of pan with half of dough.*
> *Add apple mixture to crust and dot with 2T butter.*
> *Cover with rest of dough and make slits.*
> *Bake for 45 minutes at 425 degrees.*

But the crust of an apple pie is common to many pies. So this recipe might be easier to follow if the steps specific to apple pie were kept *separate* from the details of making a crust. It would then look like this:

recipe Apple pie:

> *Peel, core and slice 8 apples.*
> *Mix together 1/3 c sugar, ¼ c flour, 1 t cinnamon in a bowl.*
> *Stir in apples.*
> ***Make pie crust dough.***
> *Line bottom of pan with half of dough.*
> *Add apple mixture to crust and dot with 2T butter.*
> *Cover with rest of dough and make slits.*
> *Bake for 45 minutes at 425 degrees.*

The details for making pie dough could be on another page:

recipe Pie crust dough:

> *Add 1/2 c butter and 5t shortening, cutting into small pieces with pastry cutter.*
> *Mix in 8t ice water.*
> *Turn the dough onto a lightly floured surface, knead it together.*

Or maybe you don't even need this crust recipe– you use some pre-made pie crust you bought at the store!

Programmers often use shortcuts like this. They rarely write entire programs from scratch by themselves. Instead, they make frequent use of pre-made code written by others. Code to be used this way is packaged into units called **functions**.

So far, when we have written our programs we have written one function, which we have named **main**. We will now examine more carefully how to have our **main** function use other functions that have been pre-made for us.

B. Functions to display output

We have seen the use of functions before. Here is a use of the **printNow** function, which prints a value on the console:

```
printNow( "Hey" )
```

You will notice that this line consists of the name of the function followed by parentheses. What sits inside the parentheses is called a **parameter**. A parameter is a piece of information that a function needs to do its job. The **printNow** function prints a value at the console, so it clearly needs to be told *what* value to print.

There is some important vocabulary that programmers use when talking about functions. The line above is said to be a **call** to the function **printNow**. We say the parameter **"Hey"** is **passed** to the function.

The data type of the parameter passed to **printNow** can be string as shown above. It doesn't matter whether the string value comes from a literal or a variable, or even from the results of some operations:

```
printNow( "Hey" )
message= "Go on"
printNow( message )
printNow( message + "!!!" )
```

The data type of the parameter passed to **printNow** can also be an integer:

```
printNow( 23 )
n= 200
printNow( n )
printNow( 2*n + 1 )
```

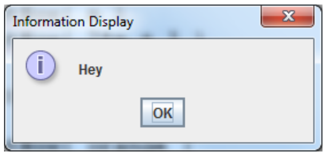
As you may recall, you can even print a color:

```
printNow( orange )
color= makeColor( 10,100,10 )
printNow( color )
```

Of course, what you see printed are the RGB values of the color, not something that actually is colored!

Another function, named **showInformation**, takes a string as a parameter and displays it in box:

```
showInformation( "Hey" )
```



Unlike **printNow**, the data type of the parameter **must** be a string. If you wish to print the value of an integer, this requires some extra work, as described later in this chapter.

Using **showInformation** is useful when you want the entire program to pause and wait for the user to click OK (or hit Enter).

☐ Practice B

B1. What is wrong with the following program?

```
def main():
  workday= 8
  hoursWorked= workday * 5
  showInformation( hoursWorked )
```

C. Functions that return values

Do you remember how the word "function" was used in your 9th grade algebra class? It was something that took a value and gave another value as a result. For example, you will remember the square root function:

$$y = \sqrt{x}$$

Python also provides a square root function, named **sqrt**. Here are some examples of its use;

```
x= 81
y= sqrt( x )
printNow( y )
printNow( sqrt( 16 ))
```

The output of this code is:

```
9
4
```

So there are functions in Python that give results just like functions in mathematics. That is, **sqrt(16)** is an expression whose value is 4. We say that we are **calling** the function **sqrt** with the **parameter** 16 and it **returns** the value 4.

The functions **printNow** and **showInformation**, by contrast, did not return anything. It would be silly to see a line of code like:

```
y= printNow( "Hey" )
```

because the expression `printNow( "Hey" )` has no value.[5] So there is nothing to assign to **y**. Calling the function **printNow** is really just a command.

It is also silly to have a line like:

```
sqrt( 81 )
```

by itself. This is not a command. It is the value 9, but that does actually not tell the computer to do anything.

You might not use the square root function very much, but here is a function that Python programmers use all the time: **str**. It takes one parameter, and returns a string that represents that value.

[5] Technically, it has the value **None**. Which is pretty useless.

For example,

```
s= str( 417 )
```

stores the string `"471"` in the variable **s**. Don't confuse the string `"471"` with the integer `471`. They are completely different.

Using **str** to turn things into strings is very useful. For example,

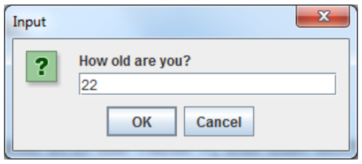
```
showInformation( 41 )
```

is an error, but the following line is fine:

```
showinformation( str( 41 ) )
```

We have seen other functions that return values. For example, here is how you get an integer entered by the user:

```
age= requestInteger( "How old are you?" )
```



The name of the function is **requestInteger**. It takes one parameter, which must be a string (the message to be displayed, "How old are you?"). It returns an integer (the value that is in the input box when the user hits OK, 22).

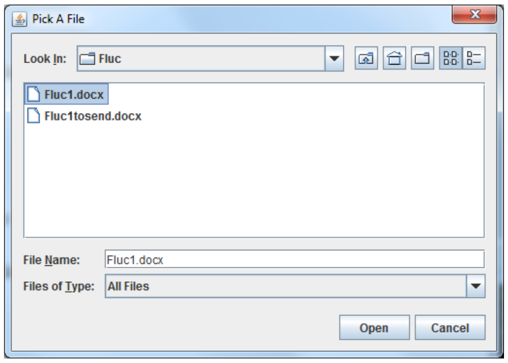
Some functions take more than one parameter. You may recall the function that makes a color from its RGB values:

```
color= makeColor( 100, 50, 20 )
```

This function takes three parameters (all integers); it returns a color.

Some functions take no parameters at all. Here is a function that allows a user to pick a file:

```
filename= pickAFile()
```

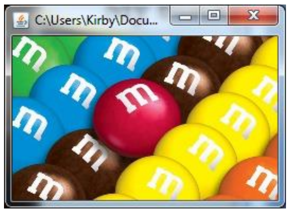


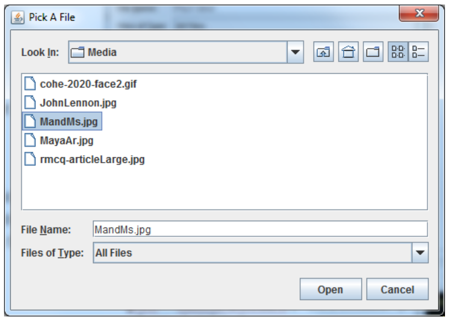
It returns a string, the full path to the file selected, such as:

```
C:\Users\Kirby\Documents\AY2010\Rw\Fluc\Fluc1.docx
```

One of the most useful applications of this function is allow the user to select a digital image. Here is an example that uses three functions:

```
filename= pickAFile()
pic= makePicture( filename )
show ( pic )
```





Look In: Media

cohe-2020-face2.gif
JohnLennon.jpg
MandMs.jpg
MayaAr.jpg
rmcq-articleLarge.jpg

File Name: MandMs.jpg

Files of Type: All Files

The function **makePicture** take one parameter (a string) and returns a picture. The **show** function takes one parameter (a picture) and returns nothing.

In summary, a function can take zero, one, two, or more parameters. It can return nothing, or return a single value.

☐ Practice C

C1. What is wrong with the following program?

```
def main():
  requestInteger("Enter your age:")
  printNow( age )
```

C2. What is the output of the following program?

```
def main():
  a= 5
  b= 20
  c= a*b + a
  message= "The answer is " + str( c ) + ", I am sure."
  printNow( message )
  showInformation( message )
```

This suggests a general technique for printing complicated messages on the console. You will use this a lot.

D. Functions in modules

Programmers have access to thousands of pre-written functions. Rather than providing one giant list of thousands of functions, programmers prefer to have them organized into groups. In Python, these groups are called modules.

For example, Python provides one module called **random**, which includes functions that pick random numbers. It also includes a module called **time**, which includes functions that deal with timing.

Many functions require programmers to request that the module be imported into their code so they can use the function. For example, there is a function called **randrange** that returns random integers in a certain range; **randrange(10,15)** returns a random integer from 10 to 14 (yes, not to 15). But if you want your program to use this function, you must include an **import** statement at the top of your file:

```
from random import *

def main():
  printNow( randrange(10,15) )
  printNow( randrange(10,15) )
  printNow( randrange(10,15) )
```

If we run this program we might get the output:

```
12
11
12
```

but if we run it again we will probably get a different sequence of three values, since, after all, they are meant to be random.

Here is an example of a program that uses the **clock** function in the **time** module. The function **clock** returns the number of seconds since the first time it was called when JES first launched.

```
from time import *

def main():
  time1= clock()
  showInformation( "Click ok when ready" )
  time2= clock()
  printNow( time2 - time1 )
```

This program prints how long I took to click the OK button on the box that popped up. (Make sure you understand how this works!)

E. A first collection of useful functions

Returns	Name	Parameters	Example
Conversion between data types			
string	**`str`**	(anything)	**`s= str( 41 )`**
Output[1]			
none	**`printNow`**	(anything)	**`printNow( 41 )`**
none	**`showInformation`**	(string)	**`showInformation("Hey")`**
Input			
integer	**`requestInteger`**	(string)	**`count= requestInteger( "How many?" )`**
string	**`requestString`**	(string)	**`name= requestString( "Who?" )`**
color	**`pickAColor`**	( )	**`myFavoriteColor= pickAColor()`**
string	**`pickAFile`**	( )	**`fileName= pickAFile()`**
Pictures			
color	**`makeColor`**	(integer,integer,integer)	**`sicklyGreen= makeColor(60,56,74)`**
picture	**`makeEmptyPicture`**	(integer,integer,color[2])	**`pic= makeEmptyPicture(100,100,black)`**
picture	**`makePicture`**	(string)	**`pic= makePicture( fileName )`**
none	**`show`**	(picture)	**`show( pic )`**
integer	**`getWidth`**	(picture)	**`width= getWidth( pic )`**
integer	**`getHeight`**	(picture)	**`height= getHeight( pic )`**
In module **random**			
integer	**`randrange`**	(integer, integer)	**`randomMonth= randrange(1,13)`**
In module **time**			
float	**`clock`**	( )	**`startTime= clock()`**

[1] There is another way to get output at the console: the **print** statement in Python. This is *not* a function; it is part of the Python language itself. You can print a number of items together by listing them separated by commas:

```
print name,age,income
```

Since this is not a function call, there are no parentheses. One downside is that, in JES, **print** statements don't actually print until your entire program has finished running. This can be confusing if you also interleave them with calls to the **printNow** function.

[2] If this third parameter is omitted, the color is white by default.

Learning Outcomes for Chapter 3

☐ Be able to explain the following terms:

- function
- function call
- parameter
- passing a parameter
- returning a value

☐ Be able to understand code that uses the functions in the table above.

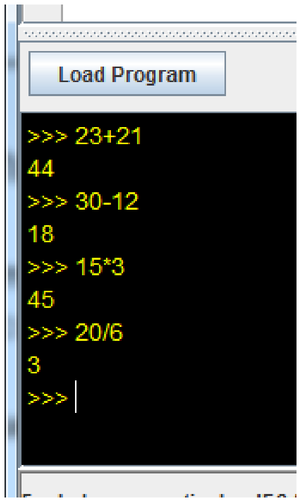
☐ Be able to write short programs that use the functions in the table above.

4 Arithmetic and Decisions

A. Arithmetic

Of all the myriad data types around, the one you will deal with the most is the humble **integer**. Integer arithmetic should be familiar to you: addition (+), subtraction (-), multiplication (*) and division (/). The technical term for these four symbols is **operator**. That is, you might hear people say "programmers use an asterisk as the multiplication operator." Specifically, you say these are **arithmetic operators**.

You can use the JES console to experiment with different arithmetic operations, using it as a calculator, as shown on the left here.



```
>>> 23+21
44
>>> 30-12
18
>>> 15*3
45
>>> 20/6
3
>>>
```

The first three operations should have no surprises, but for 20/6 did you expect the answer 3? You might say, "Ah, it did the division to get the answer 3.3333…., but then threw out the part to the right of the decimal to turn it into an integer." This is **wrong**. It uses the more basic integer division that you would have learned about in elementary school before you learned about decimals:

$$\begin{array}{r} 3\ R\ 2 \\ 6\,\overline{)\,20} \\ \underline{18} \\ 2 \end{array}$$

Here 20/3 gets you the quotient, 3. (If you want the remainder, you write 20%3, which is 2.) Everything is integers all the way.

By contrast, if the value on the left or the right of the / sign is a float, then the result will be a float. For example, 12/5 is 2, but 12.0/5 is 2.4, as is 12/5.0 and 12.0/5.0.

Arithmetic operators combine with variables and literals to make **arithmetic expressions**. Here are some arithmetic expressions: **`45`**, **`x`**, **`5*(x-1)/(y+1)`**. They often appear on the right hand side of assignment statements, for example:

```
y=  5*(x-1)/(y+1)
```

You also find arithmetic expressions in statements that print out values, such as:

```
printNow(  5*(x-1)/(y+1) )
```

In all cases, an arithmetic expression must have a definite value. In this case, this means that the variables **x** and **y** must already have values for the expression to make sense.

As you recall from 9th grade algebra, parentheses are important: $2x+9$ is different from $2(x+9)$. In Python as in algebra, * and / are performed first, and + and – are performed last. If you need to compute your answers in a different order, use parentheses. For example:

```
x= 5
y= 10
printNow( 2*x+9 )
printNow( 2*(x+9) )
```

The first line of output above is 19, the second one is 28.

☐ Practice A

A1. Write the output of the following program. Then run it in JES to check your answers.

```
def main():
  a= 10
  b=  5
  c= a + b/2
  d= (a - b)/2
  printNow( c )
  printNow( d )
  printNow( c + d*10 )
```

B. Walking through code

Variables vary. That is, their values can change as the program runs. It is important to understand how a program keeps track of variables as they change. You will do some exercises in class on this. Here we just give a tour of the idea.

Consider the following program. We show it running at the console on the right:

```
def main():
  a= 10
  printNow( a )
  a= 20
  printNow( a )
  a= a + 1
  printNow( a )
  a= a + 1
  printNow( a )
  a= a / 2
  printNow( a )
```

```
>>> main()
10
20
21
22
11
>>>
```

A computer will execute this main function in less than a hundred thousandth of a second. But sometimes we humans need to run code ourselves, using our own (slower) brains. This is called "walking through code", and can be a very important technique for debugging. You should be able to predict the output of code by walking through it statement-by-statement, making a note of how the variables change. In the code above, there is a single variable, **a**. The computer stores the value of variables in its memory. You would might use a sheet of scratch paper. When the main function just begins, a is undefined, so your scratch paper might look like this:

```
a:
```

After the first assignment statement, your scratch paper will look like this:

a: 10

After the second assignment statement, your scratch paper will look like this:

a: ~~10~~ 20

Now for the third assignment statement. It happens in two *completely separate* steps (this is **very important**):

1. The value on the right hand side is computed. This is **a+1**. What is **a**? Just consult your scratch paper above (just as the computer consults its memory): **a** is 20. So **a+1** is 21.

2. The value on the *right* hand side is assigned to the variable on the *left* hand side. That is, the value of a now becomes 21.

So now your scratch paper will look like this:

a: ~~10~~ ~~20~~ 21

After the next two assignment statements, your scratch paper will look like this:

a: ~~10~~ ~~20~~ ~~21~~ 22

a: ~~10~~ ~~20~~ ~~21~~ ~~22~~ 11

The program example above put in **printNow** commands so we could check that **a** changes just like we think.

It never makes sense to have an arithmetic expression on the left hand side. For example:

```
x+1 = 50
```

is illegal. (No, Python is not smart enough to figure out that x is 49.)

☐ Practice B

B1. Write the output of the following program. Then run it in JES to check your answers.

```
def main():
  a= 10
  b= 5
  a= b+1
  b= 2*a + 100*b
  a= 2*a + 100*b
  printNow(a)
  printNow(b)
```

C. The `if` statement

Sometimes you want to execute a sequence of statements conditionally, that is, if a certain condition is true. For that reason, every programming language has a so-called **if statement**.

The syntax in Python is straightforward. Some examples:

```
if age < 21:
  printNow( "No alcohol!" )


height= requestInteger( "How tall is your vehicle?")
bridgeClearance= 15
if height > bridgeClearance:
  printNow( "Crunch!!" )
```

The word **if** must be followed by something that can be interpreted as true or false. This is called the **condition**, or the "test." It ends with a colon, and the statements beneath it, making up what is called a **code block**, are indented.

Indentation tells us which code is part of the block (and so is executed only when the condition is true) and which code is outside it. For example, look at the following:

```
n= requestInteger( "Pick a positive number" )
if n < 0:
  printNow( "I said a positive number!" )
  printNow( "Were you listening?" )
printNow( "Ok." )
```

If the user enters the number 17, this code prints

```
Ok.
```

If the user enters the number –45, this code prints

```
I said a positive number!
Were you listening?
Ok.
```

An **if** statement must end with a colon (**:**) and must be followed by at least one indented line.

D. Comparing integers

Often in an **if** statement you want to check whether two things are equal. You might think you would use the ordinary equals sign (=) for this. But…. not quite. Look at the following examples:

```
n= requestInteger( "Pick a lucky number" )
if n == 13:
  printNow( "That is not a lucky number!" )


name= requestString( "What is your name?" )
if name == "Mildred":
  printNow( "My, what a beautiful name." )


w= requestInteger( "How wide?" )
h= requestInteger( "How tall?" )
if w == h:
  printNow( "It is square." )
```

To check whether two things are equal, you use a double equals signs (==). Remember, the single equals sign (=) is used in *assignment statements* to change the value of a variable. That is *very* different:

a==b	checks *whether* **a** is equal to **b** (true or false)
a=b	is a command to *change* the value of **a** to the value of **b**

You see assignment statements on lines by themselves, but you never see equality tests (**==**) by themselves. They are part of other statements such as **if** statements.

Besides **<**, **>**, and **==**, mentioned so far, there are other ways to compare integers:

not equal	**!=**
less than or equal	**<=**
greater than or equal	**>=**

You cannot have any blanks in between the two symbols. And their order is important. For example, you cannot say **=<**, it must be **<=**.

You can compare integer values that come from arithmetic expressions, not just simple variables or literals:

```
w= requestInteger( "How wide?" )
h= requestInteger( "How tall?" )
if w*h > 2*maxArea:
  printNow( "It is more than twice the maximum area allowed!" )
```

☐ **Practice D**

D1. Write the output of the following program. Then run it in JES to check your answers.

```
def main():
  a= 10
  b=  5
  if a <= b:
    printNow( "alpha" )
  printNow( "beta" )
  if a != b:
    printNow( "gamma" )
    printNow( "delta" )
  if b + 5 == a:
    printNow( "epsilon" )
  if b > a/2:
    printNow( "zeta" )
    printNow( "eta" )
  printNow( "theta" )
```

Learning Outcomes for Chapter 4

☐ Be able to explain the following terms:

- operator
- arithmetic expression
- condition
- code block

☐ Be able to walk through code that uses integer arithmetic and simple **if** statements.

☐ Be able to write short programs that use integer arithmetic and simple **if** statements.

5 Complex Decisions

A. Combining conditions

Sometimes in an **if** statement we want to check more than one condition. We can do this using **and**. Here's an example:

```
if age > 18  and  age < 36:
  printNow( "Dream demographic!" )

if  getWidth(pic1)==getWidth(pic2)  and  getHeight(pic1)==getHeight(pic2):
  printNow( "The two pictures are the same size." )
```

The things you combine together with **and** must be true or false. For example, the following is a syntax error:

```
if age > 18  and  < 36:
  printNow( "Dream demographic!" )
```

The left side of the **and** is fine here: **age>18** is something that can be true or false. The right hand side is the problem: "< 36" by itself is incomplete, so it can't be true or false.

You can also use an **or** between conditions:

```
if age < 15  or  age > 95:
  printNow( "You probably shouldn't be driving" )

if  getWidth(pic1)!=getWidth(pic2)  or  getHeight(pic1)!=getHeight(pic2):
  printNow( "The two pictures are NOT the same size." )
```

B. Two-way decisions

Sometimes you want to do one thing if a condition is true, and another thing if a condition is false. This is what the else statement is for in Python:

```
if age < 18:
  printNow( "You are underage." )
else:
  printNow( "You are an adult." )
```

This code is equivalent to the following:

```
if age < 18:
  printNow( "You are underage." )
if age >= 18:
  printNow( "You are an adult." )
```

but programmers would always use the first version, since it's simpler and asks only one question about the value of age, not two.

As always, in these indented blocks you can have more than one line if you like.

C. Multi-way decisions

Sometimes decisions lead to many different alternatives. This is where the **elif** statement (pronounce it "else if", which captures its meaning) comes in. Here is an example of a decision with three branches.

```
if age < 18:
  printNow( "You are underage." )
elif age >= 65:
  printNow( "You are a senior." )
else:
  printNow( "In the prime of your life, neither young nor old.")
```

In this case exactly *one* of the **printNow** statements will be reached. They are mutually exclusive alternatives.

Here is an example of a decision with five branches, with a few surrounding statements added.

```
cost= requestInteger( "How much did it cost?" )
if cost == 0:
  printNow( "free date" )
elif cost < 15:
  printNow( "cheap date" )
elif cost < 100:
  printNow( "not so cheap date" )
elif cost < 250:
  printNow( "expensive date" )
else:
  printNow( "outrageously expensive date" )
printNow( "another?")
```

Again, these are mutually exclusive alternatives. We do each true/false test in turn. When a test is true, we do the corresponding statement, and then exit the entire decision. For example, in the code above, if the user enters **75**, the code prints

```
not so cheap date
another?
```

If there is nothing to do in the final "else" case, you can just omit it altogether. For example:

```
if age < 18:
  printNow( "You are underage. " )
elif age >= 65:
  printNow( "You are a senior." )
```

Here if age is 33, there is no output.

☐ Practice C

C1. There are a couple errors in the following program. Find them.

```
def main():
  height=  requestInteger( "How tall are you in inches?" )
  if height > 40
     printNow( "You can ride this rollercoaster." )
  elif
     printNow( "Sorry, try the kiddie rides." )
```

D. Nesting

In the blocks of code inside if/elif/else statements, you can also have if/elif/else statements. This is called **nesting**. Example:

```
if age < 17:
  if rating == "R":
     printNow( "Parent or guardian please" )
  elif rating == "X":
     printNow( "Go away" )
```

☐ Practice D

D1. What is the output of the following program when the user enters 4? When the user enters 7? When the user enters 20?

```
def main():
  a= requestInteger( "Enter the value of a" )
  b=  5
  c= 10
  if a > b:
    if a > c:
      printNow( "alpha" )
      printNow( "beta" )
    elif a < c:
      printNow( "gamma" )
    printNow( "delta" )
  else:
    printNow( "epsilon" )
```

E. That was short

Only three pages of new material in this chapter? Yes. You will find as you get deeper into programming, there are fewer and fewer new parts of Python to learn.[6] Your attention gradually shifts to writing longer and longer programs. The idea is that *a very few* basic building blocks are used to build *very complicated* programs. The slides, classes, code examples, lab exercises and take-home assignments are very important here.

[6] In fact you might be interested to learn that even though you are now only a third of the way through the course, there are really only 3 more essential Python statements to learn: **for**, **while**, and **return**. Does that mean it gets easier? No….

Learning Outcomes for Chapter 5

☐ Be able to explain the following terms:

- multi-way decision
- nesting

☐ Be able to walk through code that uses **if** / **elif** / **else** statements and complex conditions with **and** and **or.**

☐ Be able to write short programs that use uses **if** / **elif** / **else** statements and complex conditions with **and** and **or.**

6 Repetition

A. How to repeat

Software allows humans to automate tedious work. One of the ways something can be tedious is by requiring a lot a **repetition**. Just as a machine in a bottling factory can put thousands of caps on soda bottles very quickly, a smart device can do millions of calculations very quickly. To achieve this kind of repetition in code, you certainly don't write a million individual lines in your program. You simply want to be able to tell your device to repeatedly execute a block of code, over and over again.

There are two kinds of repetition:

1. Repeat something while a condition is true, stopping when the condition is no longer true.

2. Repeat something a fixed number of times.

This distinction is basic, and applies to the concept of repetition in general (not just in programming). In this chapter, we concentrate on the first kind of repetition. Examples of the first kind of repetition are:

```
while you still owe money:
   keep paying

while you still have fuel in your tank:
  keep driving

while the program won't accept the password you type:
  type in another password
```

Python (like most languages) has a **while** statement that allows you to achieve this kind of repetition. The general form of a while statement is:

```
while condition:
  block of code to be repeated
```

Python repeats this block of code only *while the condition is true*. To put it another way, Python repeats this block of code *until the condition is false*. Just as with **if** statements, the condition must be something that is true or false. And just as with **if** statements, indentation is crucial to determine what statements are inside the block of code to be repeated.

Here is some code that keeps picking a random number between 1 and 10 until a seven comes up:

```
def main():
  n= randrange(1,11)
  printNow( n )

  while n!=7:
     n= randrange(1,11)
     printNow( n )

  printNow( "Finally a seven!" )
```

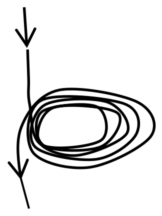
If we run this program we might get output like this:

```
4
3
9
9
5
2
8
7
Finally a seven!
```

A **while** statement followed by a block of code is called a **while loop**. Programmers call any fragment of repeating code a **loop**, and they also use the word "looping" as another way to say "repeating".

The loop metaphor comes from tracing out a path through a listing of code. When there is repetition, the path loops back on itself:

```
def main():
  n= randrange(1,11)
  printNow( n )

  while n!=7:
     n= randrange(1,11)
     printNow( n )

printNow( "Finally a seven!" )
```



☐ **Practice A**

A1. Suppose you removed two of the lines from the code above, to get this program:

```
def main():
  while n!=7:
     n= randrange(1,11)
     printNow( n )
  print "Finally a seven!"
```

What is wrong here?

B. Beware the infinite

Here is another version of the program in the previous section, looking for lucky number 13:

```
def main():
  n= randrange(1,11)
  printNow( n )

  while n!=13:
     n= randrange(1,11)
     printNow( n )

printNow( "Finally a thirteen!" )
```

The problem is that the random numbers are only between 1 and 10, so the number 13 will *never* come up. That means the condition `n!=13` will always be true. That means the block of code will repeat forever. This is called an **infinite loop**.

Infinite loops are bad.

So: never write a condition in a **while** statement that will always be true. You can do this by accident if you accidentally write a **while** statement when you meant to write an **if** statement. For example, the code:

```
age= requestInteger( "How old are you?" )
if age  < 18:
   printNow( "Sorry, adults only." )
```

is perfectly fine. But if you replaced that "if" with a "while" you would be in trouble:

```
age= requestInteger( "How old are you?" )
while age  < 18:
   printNow( "Sorry, adults only." )
```

If the user enters 23, what happens?

So be very careful with **if** and **while**. In English, we can be sloppy about the difference. The sentences

If I am walking in the forest, I am happy.
While I am walking in the forest, I am happy.

seem to mean almost the same thing. But in programming, **while** always means **repetition**. **If** does not.

C. Until

In English we also use the word "until" to describe repetitions with conditions:

```
until you no longer owe money:
   keep paying

until you have no more fuel in your tank:
   keep driving

until the program accepts the password you type:
   type in another password
```

Do you see how these statements mean the same thing as the statements that used "while" in section A? Notice how, when we change from "while" to "until", we change the condition to its *opposite*.

Although some programming languages have **until** statements, most popular languages today, like Python, do not. This is because anything you can express with an **until** you can express with a **while**, as we just saw. So instead of writing

```
until n==7:
```

which is illegal Python, you would write:

```
while n!=7:
```

Note how we reverse the condition: "until n equals 7" means "while n does *not* equal 7".

D. Complex conditions

As you learned when you studied **if** statements, conditions can be combined with **and** and **or**. Here is a little guessing game. We want the user to guess a number between 1 and 100. But we don't want the game to go on forever, so we give the user only 25 guesses. So we use a variable named **numberOfGuesses** to keep count. Here is the code:

```
target= randrange(1,101)
guess= requestInteger( "Guess the number:" )
numberOfGuesses= 1

while guess != target and numberOfGuesses < 25:
  guess= requestInteger( "Wrong. Guess again:" )
  numberOfGuesses= numberOfGuesses + 1

printNow( "Game over" )
```

This is pretty bare bones. In class we will upgrade this to something more interesting.

☐ Practice D

D1. In pseudocode, you could write the while statement above in the equivalent form:

until guess==target or numberOfGuesses >=25:

Study the difference between the "while" phrasing and the "until" phrasing. Translate the following lines of pseudocode into Python:

until count < 10 and amount > 100:

until count < 10 or amount > 100:

until count <= 10 and amount > 100 and total == 50:

Do you see a pattern?

Learning Outcomes for Chapter 6

☐ Be able to distinguish two kinds of repetition.

☐ Be able to explain the following term and give examples of them in Python:

- loop
- infinite loop

☐ Be able to walk through code that uses **while** statements that have simple or complex conditions.

☐ Be able to understand the relationship between "doing something *while*" and "doing something *until*".

☐ Be able to write short programs that use **while** statements.

7 The Simple List

A. Packaging data into lists

We live in a world of big data. Programs need to process vast amounts of information- thousands, millions, or billions of pieces of data. But if in order to process a thousand integers in our code we would need to have a thousand integer *variables*, that would be ridiculous. Instead, we can bundle up lots of values into **lists**.

Here is a list of integers, written in Python: **`[34, 90, 12, 0, 7]`**. Here is a list of floats: **`[3.24, 2.003]`**. Here is a list of strings: **`["Covington", "Newport", "Alexandria"]`**. Notice how we use **square brackets**, with the entries separated by **commas**. We will call these expressions **list literals**.

A list is a new kind of data type for us. (A list of strings is not a string!) We can assign lists to variables:

```
fruit= [ "banana", "apple", "peach", "cherry", "cantaloupe", "orange" ]
```

Order is very important in lists. The following list is very different from the previous one:

```
fruit2= [ "banana", "orange", "cherry", "cantaloupe", "apple", "peach" ]
```

although they contain the same items[7]. You can print the values of these list variables with **printNow**:

```
printNow( fruit )
```

The familiar **str** function will convert lists to strings, to be used in information boxes or complex messages:

```
showInformation( str(fruit) )
printNow( "I promise to eat 3 servings of " + str(fruit) )
```

It is perfectly legal to have a list with nothing in it. This is called the **empty list**:

```
friends= [ ]
```

And of course it is fine to have just one item in a list:

```
friends= [ "you" ]
```

Python provides a function named **len** that gives the length of a list. For example, **`len(fruit)`** is 6 and **`len(friends)`** is 0.

You don't have to use just literals to define lists. You can use expressions too:

```
n= 5
myNumbers= [ n, 10*n, 100*n+1 ]
printNow( myNumbers )
```

will print

[5, 50, 501].

[7] What do you call the things inside lists? *Elements*. *Entries*. *Items*. There is no standard term. Take your pick.

Please watch what you type:

```
        fruit3= ( "banana", "orange", "cherry", "cantaloupe", "apple", "peach" )
```

is **not** a list! You must use **square** brackets **[ ]**, not parentheses. JES will *not* give you a yellow line on this mistake however, since it is technically legal Python: it is something called a *tuple*, which we will not cover.

B. Accessing items in lists

To access items in their list you refer to them by their position. The first item in a list is said to be at position number 0. (Computer scientists start counting from zero, as you recall.) The second item in a list is said to be at position 1. And so on.[8]

In Python, you follow the name of the list by a position number in square brackets to get at an item. For example, suppose we have the assignment statement:

```
        fruit= [ "banana", "apple", "peach", "cherry", "cantaloupe", "orange" ]
```

Then `fruit[0]` is `"banana"`, `fruit[1]` is `"apple"` and `fruit[5]` is `"orange"`. Programs very often refer to items by their position number this way.

```
        printNow( fruit[3] )
        favoriteDessert= fruit[1] + " pie"
```

It is nasty error when you to try to access an item with a position number that is too large:

```
        printNow( fruit[73] )
```

This is an error: there is no position 73 in the list **fruit**. A more common slip is this:

```
        printNow( fruit[6] )
```

since you might think that, because there are six items in the list, this simply accesses the last item in the list. Nope. The last item was `fruit[5]`; we count from zero.

The number inside the brackets is sometimes called the **index** of the item. You can also call it the "position number", or "slot number". Sometimes you hear it called "subscript". Above, when we tried to access **fruit[6]**, the error message we would have gotten would read: "index out of range".

Don't confuse the two uses of square brackets:

- **Usage 1:** On their own, they enclose a complete list, a literal. For example, **[3,4,5]**, or **[2].**
- **Usage 2:** Following the name of a list, they refer to a position in the list. For example, **fruit[2].**

[8] Some people say "slot" instead of "position".

☐ **Practice B**

B1. What is the output of the following code?

```
a= 8
nums= [a, a, a, 10*a, 10+a ]
printNow( nums )
printNow( len(nums) )
printNow( nums[1] )
printNow( nums[3] )
printNow( [3] )
printNow( nums[len(nums)-1] )
printNow( nums[0]+nums[1] )
```

B2. What is the output of the following code?

```
s= "ho"
t= "hum"
words= [s, t, s+t, t+s]
printNow( words )
printNow( words[0] )
printNow( words[len(words)-1] )
printNow( words[2]+words[3])
```

B3. What is the output of the following code?

```
fruit= [ "banana", "apple", "peach", "cherry", "cantaloupe", "orange" ]
i=0
while i < len(fruit):
  printNow( fruit[i] )
  i= i + 1
```

B4. What is the output of the following code?

```
fruit= [ "banana", "apple", "peach", "cherry", "cantaloupe", "orange" ]
i=0
while i < len(fruit):
  printNow( fruit[i] )
  i= i + 2
```

C. Changing items in lists

You can use assignment statements to change an item in a list. The code fragment:

```
fruit= [ "banana", "apple", "peach", "cherry", "cantaloupe", "orange" ]
fruit[0]= "plantain"
fruit[5]= "kumquat"
printNow( fruit )
```

Will print:

```
['plantain', 'apple', 'peach', 'cherry', 'cantaloupe', 'kumquat']
```

In class you will learn how much power this capability gives to programmers. In particular, you will study the so-called **swap idiom**. Here is how you swap the items in position 2 and 3 in a list, using an extra variable named **hold**:

```
hold= fruit[2]
fruit[2]= fruit[3]
fruit[3]= hold
```

☐ **Practice C**

C1. What is the output of the following piece of code?

```
fruit= [ "banana", "apple", "peach", "cherry", "cantaloupe", "orange" ]
hold= fruit[1]
fruit[1]= fruit[2]
fruit[2]= hold
printNow( fruit )
```

C2. What is the output of the following piece of code?

```
fruit= [ "banana", "apple", "peach", "cherry", "cantaloupe", "orange" ]
fruit[1]= fruit[2]
fruit[2]= fruit[1]
printNow( fruit )
```

D. Building bigger lists

You have seen two uses of the + operator: one for integers (2+3 is 5) and one for strings ("un"+"do" is "undo"). There is a third use, joining lists together:

`[3,4,5] + [2,7]` is `[3,4,5,2,7]`
`["apple", "banana"] + ["cherry"]` is `["apple", "banana", "cherry" ]`

You can also use the * operator to join multiple copies of a list together:

`5*[1,2]` is `[1,2,1,2,1,2,1,2,1,2]`
`10*[0]` is `[0,0,0,0,0,0,0,0,0,0]`

Be careful to keep all the different uses of + and * straight.

Sometimes our programs will need to have lists of consecutive integers, such as [1,2,3,4,5,6]. If these are short, we can just type in a list literal. But we might need to have a list of consecutive integers from 1 to 10000. We certainly don't want to type all that in as a literal. So Python provides a function called **range**:

```
a= range(1,7)
```

assigns **a** to the list **[1,2,3,4,5,6]**. If we want numbers 1 through 10000, we would call **range(1,10001)**. In general, **range**(N,M) gives us consecutive integers starting at N and ending at M-1.

Warning: Don't confuse **range** with **randrange**. The first one returns a list, the second returns a number!

In Python, a given list may contain items of different types. An example would be **[7, "ewe", 3.14]**. A list can even contains lists: **[23, [20,10,1], [16], [[1]]]**. In this course, however, we will not make much use of these kinds of lists. Our lists will be **simple lists**, that is, lists in which the data type of each item is the same, and which does not contain other lists.

A list of fixed size. in which all the items have the same data type, and with items stored adjacent to each other in a computer's memory, is called an **array**. In high performance applications arrays can be more valuable, but all languages allow programmers the ability to create and manipulate lists.

☐ Practice D

D1. Given the assignment statements

```
n= 10
nx= range(2,4)
s= "hi"
sx= ["down", "town"]
```

state which of the following lines are correct Python and which are not. If they are correct, write the output they produce.

```
printNow( n + n )
printNow( nx + nx )
printNow( 2*n + n )
printNow( 2*n + 1 )
printNow( 2*nx + nx )
printNow( s + s )
printNow( s + "clown" )
printNow( sx + sx )
printNow( sx[1] + sx[1] )
printNow( sx[2] )
printNow( n[0] )
printNow( n + [1] )
printNow( nx + [1] )
```

D2. Write the output of the following piece of code:

```
nums= []
count= 1
while count < 5:
  nums= nums + [count]
  printNow( nums )
  count= count + 1
```

Learning Outcomes for Chapter 7

☐ Be able recognize and write correct list literals, and know how to access and change items in lists in Python.

☐ Be able to walk through and write simple Python programs that use simple lists.

8 Moving Through Lists

A. Repetition again

You will remember from Chapter 6 that there were two kinds of repetition: repeating something while a certain condition is true, and repeating something a fixed number of times. Chapter 6 was about the first kind of repetition, which led to **while loops**. This chapter is about the second kind.

Here are some informal examples of the second kind of repetition:

```
for each envelope in the pile:
    put a stamp on it

for each person in line:
    take the person's ticket and tear it in half

for each number you shout from 1 to 50:
    do a push-up
```

You can see that, in some sense, all of these involve a list: a list of envelopes, a list of people, a list of numbers. This kind of repetition, as Python views it, is closely tied to the idea of a list.

To repeat a block of code for each item in a list, we use a **for** statement. The **for** statement plus the block of code, taken together, is called a **for loop**. Here is an example:

```
for x in ["apple", "banana", "cherry"]:
  printNow( x )
```

will print

```
apple
banana
cherry
```

The variable named **x** is assigned to each item in the list in turn. The first time through the loop, **x** is "apple". On the second repetition, **x** is "banana". On the third repetition, **x** is "cherry". Since there are 3 items in the list, the block of code is repeated three times.

The general form of a **for** loop is:

```
for variable in list:
  block of code to be repeated
```

The list here does not have to be a list literal. It can be any expression whose value is a list:

```
shoppingCart= ["apple", "banana", "cherry"]
for x in shoppingCart:
  printNow( x )
```

prints the same thing as the first example above.

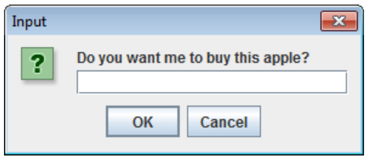
Here **x** is called the **loop variable**. Every **for** loop has a loop variable and a list. You can name the loop variable anything you want:

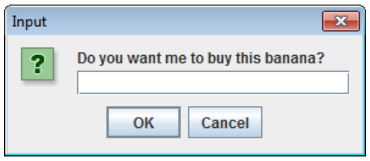
```
shoppingCart= ["apple", "banana", "cherry"]
for item in shoppingCart:
  printNow( item )
```

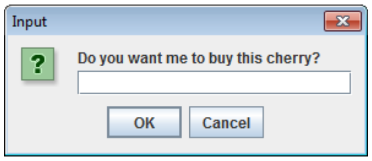
prints the same thing as the previous example.

Of course, you can do more than just one thing in a code block.

```
shoppingCart= ["apple", "banana", "cherry"]
for item in shoppingCart:
  answer= requestString( "Do you want me to buy this " + item + "?" )
  if answer=="yes":
    printNow( "OK" )
```


Do you want me to buy this apple?


Do you want me to buy this banana?


Do you want me to buy this cherry?

☐ Practice A

A1. Write the output of the following code fragment:

```
numbers= range(0,5)
for k in numbers:
  printNow( k )
```

B. Counting to yourself

How do you print "Hello there" on the console a thousand times? If you did the previous practice exercise, you probably can guess how to do it:

```
for k in range(0,1000):
  printNow( "Hello there" )
```

The variable **k** is not used in the block at all. Its sole purpose is to "keep count". (If you wanted to bang your fist on the table 27 times, almost certainly you would be counting to yourself as you do it, "*one, two, three,…*") So this example shows you how to translate the pseudocode line "*repeat the following 1000 times*" into Python.

Since we can use **for** loops with ranges to count for us, and we know how to access positions in lists, this gives us another way to print out all the elements in a list:

```
for k in range(0,len(myList)):
  printNow( myList[k] )
```

Normally you would directly work through the list the following way, which is much simpler:

```
for item in myList:
  printNow( item )
```

But it is good to be comfortable with the first way too, since sometimes you might not want to move through everything in the list. For example, this would print out the first half of the list:

```
for k in range(0,len(myList)/2):
  printNow( myList[k] )
```

☐ Practice B

B1. Write the output of the following code fragment:

```
myList= ["alpha", "beta", "gamma", "delta", "epsilon" ]
n= len(myList)
for k in range(1,n+1):
  printNow( myList[n-k] )
```

C. The "initialize and update" pattern

How do you add up all the numbers in a list? Here's one way:

```
total=0
for number in listOfNumbers:
  total= total + number
printNow( total )
```

This pattern is called a **running total**. You start out with the total being zero, before you begin. Then you go through the list, item by item, adding the item to the total. By the time you are done with the loop, you've moved through the whole list and have the grand total.

This pattern is so popular, many programming languages have introduced an abbreviation for adding something to a variable. That is, instead of writing `total= total + number`, you can use the more compact form:

```
total += number
```

Programmers say this statement "increments the total".

In a similar vein, there is a pattern called a **running maximum**.

```
max= listOfNumbers[0]
for k in range(1,len(listOfNumbers)):
   if listOfNumbers[k] > max:
      max= listOfNumbers[k]
printNow( max )
```

This finds the largest number in the list. You start out with your "guess" of the maximum so far, which at first is just the first item. Then you go through the list. Whenever you find an item bigger than the maximum so far, you reset the maximum to that item. By the time you are done with the loop, you've examined everything, and the "maximum so far" is the true maximum.

Both of these patterns involve:

1. Setting a variable to an initial value *before* the loop begins (you call this "initializing" the variable).
2. Updating the variable *inside* the loop.

☐ **Practice C**

C1. Change the example above to compute the **minimum** of a list of numbers.

C2. Explain in English what the following code fragment does:

```
count=0
for number in listOfNumbers:
  if number == 0:
       count += 1
printNow( count )
```

D. Preview of administrative scripting

One common use of repetition is when system administrators (people who manage computers and networks for groups of users) need to issue multiple commands. They write programs called "administrative scripts". But even ordinary users may want to issue a long series of repetitive commands at a console.

Here's a simple example to illustrate the idea. Suppose you want to copy one thousand files to a folder called backup, like this:

```
copy image1.jpg backup
copy image2.jpg backup
...
copy image1000.jpg backup
```

You would not type one thousand commands, of course. You would run a script that would execute those commands using a loop. This is how you would print these commands:

```
for j in range(1,1001):
   printNow( "copy image" + str(j) + ".jpg  backup" )
```

In an actual script, instead of **printNow** you would call a function (typically named **eval**) that would actually perform the command on the computer.

Learning Outcomes for Chapter 8

☐ Be able to walk through and write simple Python programs that use **for** loops.

☐ Be able to recognize and write code that uses the "initialize and update" pattern.

9 Digital Images

A. Where we start

In the first half of this course you learned about digital images, called **pictures** in JES. In addition to the picture data type, you learned about the data types **pixel** and **color**. In this chapter we use the power of **for** loops, which you learned about in Chapter 7, to do what is called "image processing."

In Chapter 3, you learned about the following useful functions:

```
pickAColor    pickAFile    makePicture         getWidth
makeColor                  makeEmptyPicture    getHeight
                           show
```

The slides for this course have introduced a few other picture functions as well (and your teacher may have introduced even more). But here we will assume only assume you are familiar with the ones above.

Digital images are stored in many different ways. You can recognize this by noting all the different types of files that hold images: JPEG, GIF, TIFF, RAW, BMP to name a few. These files hold more information than just the picture. For photos they may store the date and time when the photo was taken and even the make of the camera that took the photo.

B. Pixels

Despite the huge variety of ways to store digital images, the JES functions allow our programs to treat them all the same way: as a rectangular grid of colored **pixels**. A pixel ("picture element") is a colored dot. It has a color and a location in the picture.

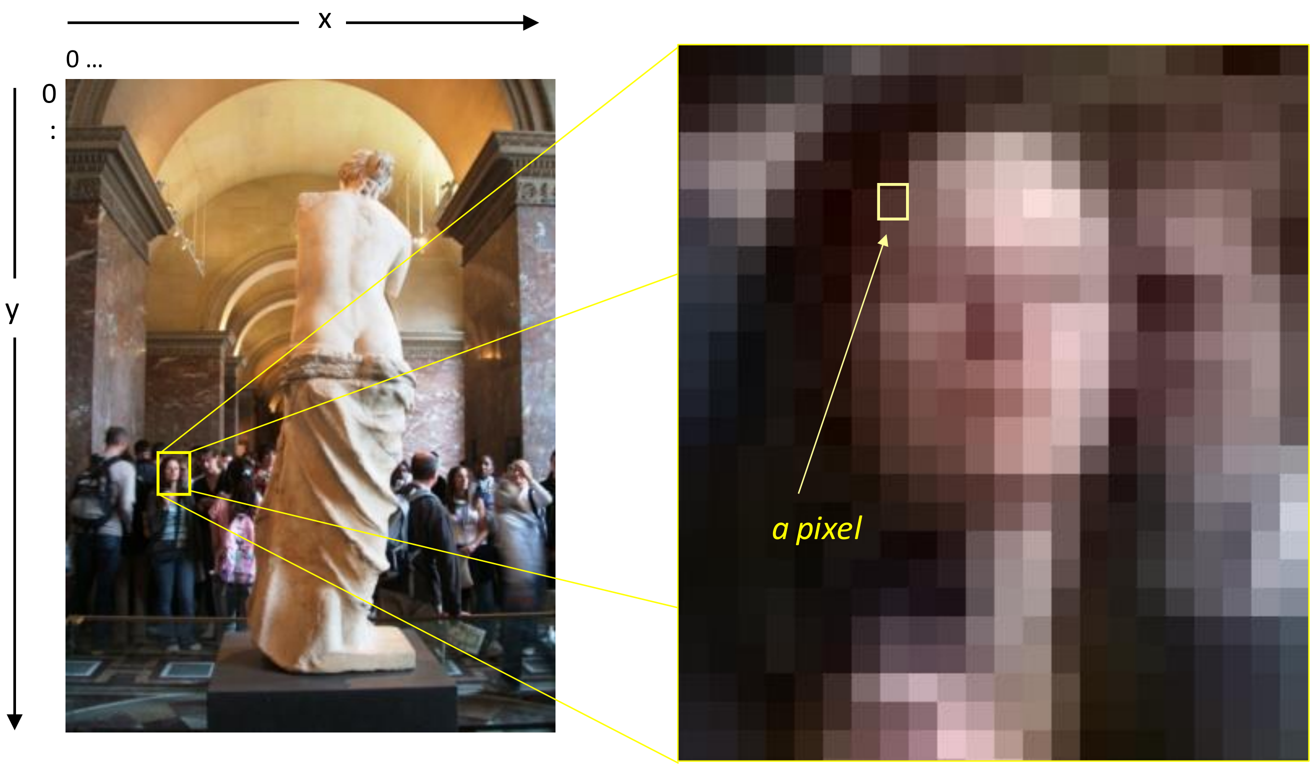
We specify a location in a picture by its horizontal position (left-to-right), and its vertical position (top to bottom). We count these positions using integers starting from 0. Following standard terminology from mathematics, it is common to use a variable named **x** for the horizontal position, and a variable named **y** for the vertical position.

You can get a single pixel from an image by calling **getPixelAt(pic, x, y)**. The first parameter is the picture you are interested in; **x** and **y** are the horizontal and vertical positions.

If you want to get the color of a pixel, you use the function **getColor(px)**; the one parameter is the pixel you are interested in. You can set the color if a pixel by using **setColor(px, color)**.

Sometimes it's convenient to directly get and set the RGB levels of a picture. JES provides a set of functions **getRed(px)** , **getGreen(px)** , **getBlue(px)** that return integers from 0 to 255 indicating the level of red, green and blue, respectively. You can also set the RGB levels using **setRed(px, level)** , **setGreen(px, level)**, and **setBlue(px, level)**, where level is an integer from 0 to 255.

Finally, there are two functions that return integers that give the horizontal and vertical positions of the a pixel: **getX(px)** and **getY(px)**. See the illustration on the next page for a summary (and also review the slides from class, of course).



Returns	Name	Parameters	Example
Input functions used often with pictures			
string	`pickAFile`	( )	`fileName= pickAFile()`
color	`pickAColor`	( )	`myFavoriteColor= pickAColor()`
Color			
color	`makeColor`	(integer,integer,integer)	`sicklyGreen= makeColor(60,56,74)`
Functions dealing with pictures as a whole			
picture	`makeEmptyPicture`	(integer,integer,color)	`pic= makeEmptyPicture(100,100,black)`
picture	`makePicture`	(string)	`pic= makePicture( fileName )`
none	`show`	(picture)	`show( pic )`
none	`repaint`	(picture)	`repaint( pic )`
integer	`getWidth`	(picture)	`width= getWidth( pic )`
integer	`getHeight`	(picture)	`height= getHeight( pic )`
list of pixels	`getPixels`	(picture)	`for px in getPixels( pic ):`
Functions dealing with individual pixels			
pixel	`getPixelAt`	(picture, integer, integer)	`px= getPixelAt( pic, x, y )`
color	`getColor`	(pixel)	`getColor( px )`
none	`setColor`	(pixel, color)	`setColor( px, yellow )`
integer	`getRed`	(pixel)	`r= getRed( px )  --` ***similarly for green and blue***
none	`setRed`	(pixel, integer)	`setRed( px, 0 ) --` ***similarly for green and blue***
integer	`getX`	(pixel)	`x= getX( px ) --` ***similarly for y***

Figure 9.1: Summary of picture-related functions in JES.
See slides and JES Help for more information.

As you know by now, there is no point in memorizing the details of these functions. When you write programs that need them, you will have access to the JES Help pages (and this textbooklet and the class slides of course). On exams you will have these summarized on a "cheat sheet". Programmers may use thousands of functions, and will always be looking them up. The point is to understand what you want to do and how to seek the functions you need to do the job.

Here is a program that puts a black dot in the middle of a picture.

```
def main():
  pic= makePicture( pickAFile() )
  show( pic )
  showInformation( "Going to put a black dot in the middle now..." )
  xMiddle= getWidth( pic ) / 2
  yMiddle= getHeight( pic ) / 2
  px= getPixelAt( pic, xMiddle, yMiddle )
  setColor( px, black )
  repaint( pic )
```

Note how we use **show** to display the picture for the first time. When we make a change to the picture and want to update it the display to reflect the change, we use **repaint**.

☐ Practice B

B1. What does the following program do?

```
def main():
  pic= makePicture( pickAFile() )
  w= getWidth( pic )
  h= getHeight( pic )
  pxA= getPixelAt( pic, 0, 0 )
  pxB= getPixelAt( pic, w-1, h-1 )
  colorA= getColor( pxA )
  setColor( pxB, colorA )
  show( pic )
```

C. Looping over all the pixels

Changing a single pixel is not very interesting. It is barely visible in an image with a lot of pixels. Instead, we want a way to repeat an operation for *all* pixels in a picture. This means we will need to get a *list* of all the pixels in a picture. The function **getPixels** does this; it is highlighted in the table on the previous page.

The assignment statement:

```
allPixels= getPixels( pic )
```

assigns a giant list of all the pixels in **pic** to the variable **allPixels**. Suppose we wanted to set all the pixels in an image to black. You could do it this way:

```
for px in allPixels:
  setColor( px, black )
```

If this seems mysterious, please review the introduction to **for** loops in the previous chapter. This **for** loop assigns the loop variable **px** to each element of the list **allPixels** in turn. So **px** refers to each pixel in the picture in turn. If there are 10,000 pixels, the call to **setColor** is repeated 10,000 times, once for each pixel.

It is much more common not to store the list of all pixels in a separate variable, but just to put the call to **getPixels** in the **for** statement directly:

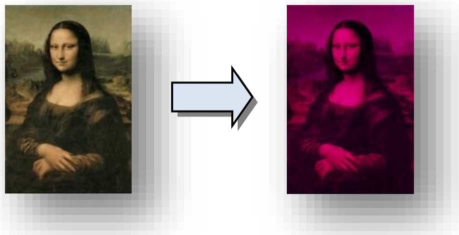
```
for px in getPixels( pic ):
  setColor( px, white )
```

This is exactly the same as the previous example.

Warning: Don't accidentally type **getPixel** when you mean **getPixels**. (The function **getPixel** is an earlier name in JES for **getPixelAt**. It only gets a single pixel, not the full list.)

Here is another example: we set the green level of each pixel in an image to zero:

```
def main():
  pic= makePicture( pickAFile() )
  show( pic )
  for px in getPixels( pic ):
    setGreen( px, 0 )
  repaint( pic )
```

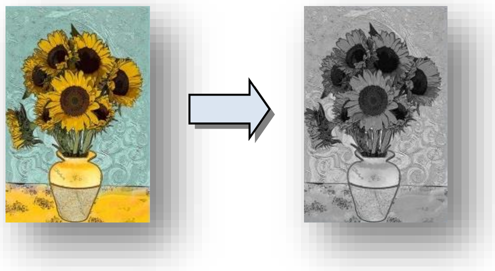


Here is a more useful example. We can think of the average of the red, green, and blue values of pixel as its **brightness** or intensity. For example, a white pixel has RGB values (255,255,255) and has brightness level 255. A red pixel (255,0,0) has brightness level (255+0+0)/3 = 85, as does a green pixel (0,255,0). What would happen if we set the RGB values of each pixel to the average of its RGB values? Here is the pseudocode:

```
for each pixel in the picture:
  store the RGB values of that pixel in the variable r,g and b, respectively
  define the brightness as the average of r,g, and b
  set the RGB values of the pixel to this brightness value
```

In Python, we get the program:

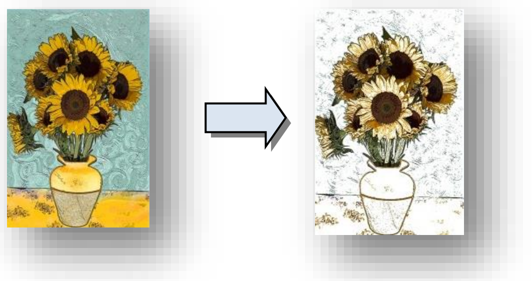
```
def main():
  pic= makePicture( pickAFile() )
  show( pic )
  for px in getPixels( pic ):
    r= getRed( px )
    g= getGreen( px )
    b= getBlue( px )
    brightness= (r+g+b)/3
    setRed( px, brightness )
    setGreen( px, brightness )
    setBlue( px, brightness )
  repaint( pic )
```



The picture has been converted to black-and-white. (The technical term is "grayscale", since its pixels are not just black or white, but can be any of 256 shades of gray.)

As another example, here we set all the *bright* pixels (meaning the brightness level is above 128, which is halfway to the maximum level of 255) to solid white:

```
for px in getPixels( pic ):
  r= getRed( px )
  g= getGreen( px )
  b= getBlue( px )
  brightness= (r+g+b)/3
  if brightness > 128 :
    setColor( px, white )
```



At this point, you should experiment on your own, using everything you have learned in the first 9 weeks of this course to try different effects on images. In class you also may have learned how to draw shapes such as lines and rectangles on pictures. You should feel free to use those functions as well, to achieve interesting visual effects. *You will learn by playing.*

☐ Practice C

C1. Find the bug in the following program.

```
def main():
  pic= makePicture( pickAFile() )
  for px in getPixels( pic ):
    setRed( pic, 0 )
  show( pic )
```

C2. Describe in visual terms what the following program does.

```
def main():
  pic= makePicture( pickAFile() )
  for px in getPixels( pic ):
    setRed( px, getRed(px)/2 )
    setGreen( px, getGreen(px)/2 )
    setBlue( px, getBlue(px)/2 )
  show( pic )
```

Learning Outcomes for Chapter 9

☐ Know how to use the JES functions to set the color values of individual pixels in an image.

☐ Know how to use the JES functions to apply changes to all the pixels in an image to achieve a certain visual effect.

10 Loops Within Loops

A. Nesting

If, **elif**, **else**, **while** and **for** statements are all followed by an indented block of code. That block of code itself can contain further **if**, **elif**, **else**, **while** and **for** statements with their own code blocks. This is called nesting. Programmers use indentation to indicate the level of nesting (and in fact Python *forces* programmers to indent).

When humans walk through nested code, it is easy to get lost. This applies in particular in the case of loops within loops. Yet these are quite common, even in non-programming situations. For example, you might tell someone, "Inspect every tire on every vehicle in the parking garage". You could express this a bit more precisely this way:

```
for each vehicle in the parking garage:
    for each tire on that vehicle:
        inspect that tire.
```

Making this more Python-like, using loop variables:

```
for each vehicle v in parking garage:
   for each tire t in the set of v's tires:
      inspect t.
```

Or even in pretend Python:

```
for v in getVehiclesIn( parkingGarage ):
   for t in getTiresOf( v ):
     inspect( t )
```

Let's explore this kind of nesting in actual Python code. Here is a piece of code that prints the numbers 1 through 4:

```
for num in range(1,5):
   printNow( num )
```

Suppose we want to repeat this three times. We already know how to repeat something three times:

```
for count in range(0,3):
   [whatever we want to repeat]
```

So what goes in this little block is exactly our code fragment that counts 1 to 4:

```
for count in range(0,3):
  for num in range(1,5):
     printNow( num )
```

We call the second **for** loop the "inner" for loop. The first one is called the "outer" **for** loop. The inner for loop prints the numbers 1 through 4. The outer **for** loop repeats the inner **for** loop three times.

The output will be:

```
1
2
3
4
1
2
3
4
1
2
3
4
```

It is not unusual for ranges in an inner **for** loop to depend on the variable in the outer **for** loop. Look at the following example:

```
for count in range(0,4):
  for num in range(count,4):
     printNow( num )
```

The *first* time through the outer loop, **count** will be 0, so the inner loop will be **for num in range(0,4).**
The *second* time through the outer loop, **count** will be 1, so the inner loop will be **for num in range(1,4).**
The *third* time through the outer loop, **count** will be 2, so the inner loop will be **for num in range(2,4).**
The *fourth* (last) time through the outer loop, **count** will be 3, so the inner loop will be **for num in range(3,4).**

The output will be:

```
0
1
2
3
1
2
3
2
3
3
```

Make sure you can walk through this code and get this output.

☐ Practice A

A1. Write the output of the following piece of code:

```
for i in range(0,3):
  for j in range(0,2):
     printNow( str(i) + " " + str(j) )
  printNow( "next" )
```

A2. Write the output of the following piece of code:

```
for count in range(0,5):
  for num in range(0,count):
     printNow( num )
```

B. Applications of nesting to images

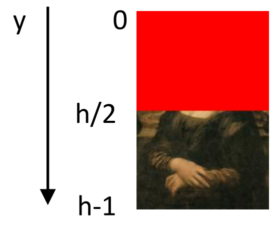
One of the limitations to the kind of image processing examples we did in the previous chapter was that we did the same thing to each pixel. All our examples were of the form:

```
for each pixel in the picture:
    do something to that pixel
```

But what if we want to only work on *some* of the pixels in the image? For example, let's say we want to make all of the pixels in the top half of the image red.

Here is one way to do that:

```
h= getHeight(pic)
for px in getPixels( pic ):
    if getY(px) < h/2:
      setColor( px, red )
```



But this is quite wasteful. We go through *all* the pixels in the image, but for half of them we know we are not going to do anything. So why bother to look at the bottom half of the picture at all?

Here is a sketch of a more efficient approach:

```
h= getHeight(pic)
for y in range(0,h/2):
   Set all the pixels in row y to red.
```

So this sets all the pixels in row 0 to red, then sets all the pixels in row 1 to red, then sets all the pixels in row 2 to red, and so on, until we reach row **h/2**, which is halfway down the image.

What remains is to translate that one line of pseudocode *Set all the pixels in row* **y** *to red* into actual Python. Well, to across row y, we let the horizontal position **x** move from 0 to **w**, where **w** is the width of the image:

```
for x in range(0,w):
   px= getPixelAt( pic, x, y )
   setColor( px, red )
```

Putting these peices together gives:

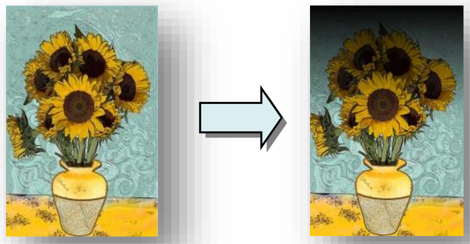
```
w= getWidth( pic )
h= getHeight(pic)
for y in range(0,h/2):
  # Set all the pixels in row y to red
  for x in range(0,w):
    px= getPixelAt( pic, x, y )
    setColor( px, red )
```

This approach with nested loops is more efficient than our original approach. (In fact, you know how to use the **time** function. Time it and see if it really is faster!)

In sum, when you process images you will have to decide between "looping through a list of all the pixels", or "looping through all the x and y positions". The latter is a nested loop.

As another example, here is how we would have the top half of an image fade to black. The idea is that we darken an RGB level by multiplying it by a real number between 0 and 1. The closer to 0 it is, the more it darkens the color. In the first row, we multiply the RGB levels by 0, making them all zeros (which yields black). Halfway down, in row **y**=**h**/2, we multiply by 1, which does not change the levels at all (no darkening). And this fade level changes linearly in between. Here is the code:

```
pic= makePicture( pickAFile() )
show( pic )
w= getWidth( pic )
h= getHeight(pic)
for y in range(0,h/2):
  #  Set the fade level proportional to the distance down from 0 to h/2.
  fadeLevel= y/(h/2.0)
  for x in range(0,w):
    px= getPixelAt( pic, x, y )
    r= fadeLevel * getRed( px )
    g= fadeLevel * getGreen( px )
    b= fadeLevel * getBlue( px )
    fadedColor= makeColor( r,g,b )
    setColor( px, fadedColor )
repaint( pic )
```



☐ Practice B

B1. Describe in visual terms what the following piece of code does to **pic**.

```
w= getWidth( pic )
h= getHeight(pic)
for y in range(0,h):
  for x in range(0,w/2):
    px= getPixelAt( pic, x, y )
    setColor( px, red )
```

B2. Describe in visual terms what the following piece of code does to **pic**.

```
w= getWidth( pic )
h= getHeight(pic)
for y in range(0,h/2):
  for x in range(0,w):
    pxA= getPixel( pic, x, y )
    pxB= getPixel( pic, x, y+h/2 )
    setColor( pxB, getColor( pxA ) )
```

Learning Outcomes for Chapter 10

☐ Be able to walk through nested code.

☐ Be able to transform images by looping over all pixels and by looping over x and y pixel locations.

11 Functions: Breaking Things Up

A. Keeping Organized

In Chapter 3 you learned to call functions. Functions help ease your burden as a programmer, allowing you to rely on the work of others. So far, you have been writing only one function per program, which you have usually called **main**. But it is easy to write other functions with other names, and you can call your own functions. Here is an example:

```
def main():
  drawAPicture():
  writeAPoem()
  showInformation( "That's all" )

def drawAPicture():
# Draws a little picture.
  pic= makeEmptyPicture( 100, 100, yellow )
  addLine( pic, 50,50, 75,75 )
  show( pic )

def writeAPoem():
# Displays the first lines of a middle English poem
  printNow( "The lyf so short, the craft so long to lerne" )
  printNow( "Th'assai so hard, so sharpe the conquerynge" )
```

The program above consists of three functions. At the console in JES, we typically type:

```
main()
```

This would call **main**. When **main** runs, you can see it first calls **drawAPicture** and then calls **writeAPoem**, before printing "That's all." When a function finishes, we say it **returns** to the point where it was called. Here, when **writeAPoem** finishes, it returns to **main**, and **main** moves on to call **showInformation**.

It is important not to get lost when calling functions. Look at the following program:

```
def main():
  alpha()
  beta()
  beta()

def alpha():
  beta()
  printNow( "I am alpha" )
  beta()

def beta():
  printNow( "I am beta")
```

When we run **main** at the console, the output would be:

```
I am beta
I am alpha
I am beta
I am beta
I am beta
```

Don't read any further until you've walked through this code and confirmed that this is indeed the output!

At the console, we could call any of these functions directly if we wanted to. For example we could type

```
alpha()
```

at the console, and the output would be:

```
I am beta
I am alpha
I am beta
```

When defining more than one function in a file, the order does not matter. For example, one could put the definition of alpha and beta first, followed by the definition of main. It is normal to insert one or two blank lines between function definitions, to make the separation clear.

☐ Practice A

A1. Consider the following program:

```
def gamma():
  printNow( "I am gamma" )
  delta()
  epsilon()

def delta():
  epsilon()
  printNow( "I am delta" )

def epsilon():
  printNow( "I am epsilon" )

def main():
  gamma()
  delta()
```

What is the output when you type **delta()** at the console?

A2. What is the output when you type **gamma()** at the console?

A3. What is the output when you type **main()** at the console?

B. Functions are islands unto themselves

"Good fences make good neighbors." This is a 17th century proverb. If you haven't thought about this proverb in a while, think about it again. It is a clue to this section.

Here is a simple little program:

```
def main():
  name= "Billie"
  showInformation( "Hello there, " + name )
```

Now, you could try to break it up this way:

```
def main():
  name= "Billie"
  greet()

def greet():
  showInformation( "Hello there, " + name )
```

But when you run this, you will get a yellow highlight on the last line, and an error message indicating that the variable name is not found.

Not found? But it's *right there*: **main** assigns the value "Billie" to name just before it gets to the greeting. What's the problem?

Well, the problem is that **each function has its own variables, and does not share them**. The variable **name** is defined within the function **main**. It is unknown to any other function. If **greet** wants to use variables, it has to define its own.

This is called "**scope**" in programming. We say that the line highlighted in yellow above is *outside the scope* of the variable **name.** Think of each function as a separate, isolated, island. Functions do not know about each others' variables!

Too see this another way, look at the following code:

```
def main():
  n= 100
  myFunction()
  printNow( n )

def myFunction():
  n= 999
  printNow( n )
```

The output is:

```
999
100
```

Follow this carefully! Did you think the output would be 999 twice? No. That's because the two variables named **n** are completely different, totally unrelated. There's **n** in **main** and there's **n** in **myFunction**. The fact that these two variables are spelled the same (namely, **n**) is a coincidence. They are no more related than Paris, Texas and Paris, Kentucky!

Why would anyone design a programming language to work like this? We'll talk about this in class.

☐ Practice B

B1. Write the output that results when you call **main** below:

```
def main():
  n= 5
  doubleMe( n )
  printNow( n )

  friends= [ "Angelina", "Brad" ]
  morePals( friends )
  printNow( friends )


def doubleMe( n ) :
  n= 2 * n
  printNow( "Inside doubleMe " + str(n) )


def morePals( friends ):
  friends= friends + ["Zorg"]
  printNow( "Inside morePals: " + str(friends) )
```

C. Passing data to a function

If the variables defined in one function are completely distinct from the variables in another function, how can two functions share data? How can one function send data to another?

We've already seen how this is done, through functions that are provided by JES. For example:

```
showInformation( "Hello there" )
```

The idea is to send information as a parameter. Here "Hello there" is a string which is being sent as a parameter to the function **showInformation**. The function **showInformation** needs this data: how else would it know what to write inside that dialog box?

Here is how we write a function that takes a parameter:

```
def main():
  apologize( 1 )
  printNow( "Still not feeling better?" )
  apologize( 5 )
  printNow( "My, you are needy" )
  apologize( 1000 )


def apologize( count ):
# Repeats an apology a given number of times.
  for i in range(0,count):
    printNow( "I'm sorry." )
```

What is the value of **count** inside the function **apologize**? Well, it's called three times by **main**, and has a different value each time. The first time **main** calls **apologize**, **count** has the value 1. The second time it has the value 5, the third time 1000.

Notice there is no assignment statement for **count** inside **apologize**, but it is still defined. That's because **count** is a parameter. **It gets its value from the place it is called**, namely back in **main**.

As you know, functions can have more than one parameter. What is the output of the following program?

```
def main():
  repeatMessage( "I'm sorry", 1 )
  repeatMessage( "I said I'm sorry", 5 )
  repeatMessage( "Enough is enough", 1 )


def repeatMessage( message, count ):
# Repeats a given message a given number of times.
  for i in range(0,count):
    printNow( message )
```

Using parameters to send information from one function to another is a deep idea. But you should make a distinction between:

- The **values** that are sent by a function to another function.
- The **names** of the parameters that receive those values when the function is called.

For example, consider the function:

```
def myFunction( a, b ):
 printNow( "a is " + str(a) +  " and b is " + str(b) )
```

At the console if you type

```
myFunction( 7, 3 )
```

you will see the output

```
a is 7 and b is 3
```

You can also call it from **main** in a variety of ways. Anything goes, as long as you send two values down to **myFunction**:

```
def main():
  myFunction( 7, 3 )
  myFunction( 10, 20 )
  myFunction( 1+1, 2+2 )
  c= 99
  myFunction( c, 10*c )
```

The output when you call **main** is:

```
a is 7 and b is 3
a is 10 and b is 20
a is 2 and b is 4
a is 99 and b is 990
```

Beware: a lot of beginners get thrown by the following situation:

```
def main():
  a= 44
  b= 77
  myFunction( b, a )
```

The output is

```
a is 77 and b is 44
```

If this surprises you, remember two things:

- Scope of variables: Functions are islands to themselves. They don't know about each others' variables.
- Parameters are passed in order: The first value in a call gets sent down as the first parameter, the second value to the second parameter, etc.

In this specific example:

1. The function **myFunction** knows nothing about the variables named **a** and **b** inside **main**.

2. The first value in the list of parameters when the function gets called, 77, gets sent down to the first parameter (named **a** in **myFunction**) and the second value in the list, 44, gets sent down to the second parameter (named **b** in **myFunction**).

As the course proceeds, we will see how incredibly powerful it is to break up a program into functions that take parameters. This is a truly important idea.

The bottom line is that now we can write more complex programs. If we have 200 lines of work to do, we can break up our work into smaller, more easily comprehensible function definitions (maybe 10-30 lines each). This will make tackling larger projects more manageable. And we will even be able to reuse some of our functions in multiple programs.

☐ **Practice C**

C1. Write the output that results when you call **main** below.

```
def zeta( a, b, c ):
  printNow( a )
  printNow( b )
  printNow( c )

def theta( a ):
  printNow( 10*a )

def main():
  a= 5
  b= 2
  zeta( a, b, b )
  zeta( a, a, a )
  zeta( a+b, a-b, 0 )
  theta( 10*a )
  theta( 10*b )
```

Learning Outcomes for Chapter 10

☐ Know how to define multiple functions in one file, and be able to walk through code in which functions call other functions.

☐ Understand the concept of scope.

☐ Understand how parameters are passed when functions are called.

12 Returning Results

A. The `return` statement

This would be a good time to review section C in Chapter 3. There you learned about the distinction between functions that are called like commands (like **printNow** or **setRed**) and functions that return values (like **requestInteger** or **getRed**).

In the previous chapter, you learned how to write your own functions that are called like commands. In this chapter, we write functions that return values.

First, let's write a function that takes one parameter, and returns a value. (This kind of function may remind you of when you first learned the word "function" in a math class.) Here is a function that takes a number and triples it:

```
def triple( x ):
  return 3*x
```

Note the new statement we have introduced here: the **return** statement. It consists of the word **return**, followed by a single value. When the function reaches this line, it takes the value and returns it to the point where the function was called.

Here is how we could call the function:

```
printNow( triple( 10 ) )
printNow( triple( 2 ) + triple( 3 ) )
printNow( triple( triple( 5 ) ) )
```

The output would be

```
30
15
45
```

Next, we write a function that takes no parameters, and returns a value. Let's write **rollDice**, a function that simulates rolling two dice. It returns the number that comes up, which will be between 2 and 12.

```
def rollDice():
  die1= randrange(1,7)  # the number that comes up on the first die, 1 thru 6
  die2= randrange(1,7)  # the number that comes up on the second die, 1 thru 6
  return die1 + die2
```

We could use the **rollDice** function this way:

```
printNow( rollDice() )
printNow( rollDice() )
printNow( rollDice() )
```

In each call, value of the expression `rollDice()` is replaced by the value returned by the function. Here is a bit of code that keeps rolling dice until a 2 comes up ("snake-eyes"):

```
count= 1
while rollDice() != 2:
  count= count + 1
printNow( "Had to roll the dice " + str(count) + " times before getting a 2." )
```

We should emphasize that after the function hits the **return** statement, no further lines of code in the function are reached. For example, if you were to add a **printNow** call at the end:

```
def rollDice():
  die1= randrange(1,7)  # the number that comes up on the first die, 1 thru 6
  die2= randrange(1,7)  # the number that comes up on the second die, 1 thru 6
  return die1 + die2
  printNow( "Hello? Hello?" )
```

that line would never be reached. It is totally pointless.

Here is another way to write the **rollDice** function:

```
def rollDice( ):
  # Returns the result of rolling two dice.
  return randrange(1,7) + randrange(1,7)
```

Notice that it begins with a comment describing what the function does; this is a good thing to do.

☐ Practice A

A1. Write the output of the function main below

```
def main():
  printNow( doubt( "this makes sense" ) )
  printNow( doubt( doubt ( "I doubt" ) ) )

def doubt( sentence ):
  return "I doubt that " + sentence
```

A2. Consider the following alternative ways to write the **rollDice** function. They are legal Python, but they won't work correctly. Explain.

```
def rollDice():
  # Returns the result of rolling two dice.
  return 2 * randrange(1,7)

def rollDice():
  # Returns the result of rolling two dice.
  return randrange(2,13)
```

B. Functions and digital images

The *picture* is a complicated data type, not like a simple number, string, or list. It is one of those data types that can be changed when it is passed as a parameter. Compare the effect of the following function when called on a string and on a picture:

```
def blackout( string, picture ):
  string= "BLACK"
  for px in getPixels( picture ):
    setColor( px, black )
```

Here is an example:

```
def main():
  s= "hello"
  pic= makePicture( pickAFile() )

  printNow( s )
  show( pic )

  blackout( s, pic )

  printNow( s )
  repaint( pic )
```

The string that is printed out after the call to **blackout** is still "Hello"; **blackout** does not change the value of the variable **s** in **main**. On the other hand, the picture that is shown in the very last line is solid black. The blackout function *has* changed the picture parameter!

What is the difference? You will learn about this in detail if you continue your study of programming beyond this course, but here is a simple rule of thumb. If you can pass a data type that could be a literal (like 23, "hello", or [10,20,30]), then the function cannot change that parameter. If there is no literal for it, the function *may* be able to change it. Whether it actually does depends on the specifics of the data type. The picture data type does allow such change.

Knowing this, and knowing how to return values, many of our digital image processing programs can be streamlined by the use of functions. The goal of writing functions is to make functions look simpler. And simpler is better.

For example, compare and contrast the following two programs that ask a user to enter a picture, then displays it in black and white.

Here is the way we would have written it before knowing what we know from Chapters 11-12:

```
def main1():
  pic= makePicture( pickAFile() )
  show( pic )
  for px in getPixels( pic ):
    r= getRed( px )
    g= getGreen( px )
    b= getBlue( px )
    brightness= (r+g+b)/3
    grayLevel= makeColor( brightness, brightness, brightness )
    setColor( px, grayLevel )
  repaint( pic )
```

Here is the way we can write it by breaking things up into functions:

```
def main2():
  pic= makePicture( pickAFile() )
  show( pic )
  convertToBW( pic )
  repaint( pic )

def convertToBW( pic ):
  for px in getPixels( pic ):
     brightness= getBrightness( px )
     grayLevel= makeColor( brightness, brightness, brightness )
     setColor( px, grayLevel )

def getBrightness( pixel ) :
   r= getRed( pixel )
   g= getGreen( pixel )
   b= getBlue( pixel )
   return (r+g+b)/3
```

The total number of lines here is longer, but most programmers would consider this a much better approach. It is easier to follow. In reading **convertToBW** we don't need to get bogged down in the details of calculating brightness. In **main2**, we can just see a high level outline of what the program does, a kind of "executive summary." Further, you can imagine working on the details of **getBrightness** and **convertToBW** separately. You can test them separately. And you can reuse them in future programs.

This is one of the deepest ideas in software development. The fancy term for it is *behavioral decomposition*: breaking up complicated behaviors into separate simpler behaviors.

Learning Outcomes for Chapter 12

☐ Know how to define function that return values.

☐ Understand how programs can be made simpler by breaking them up into functions.

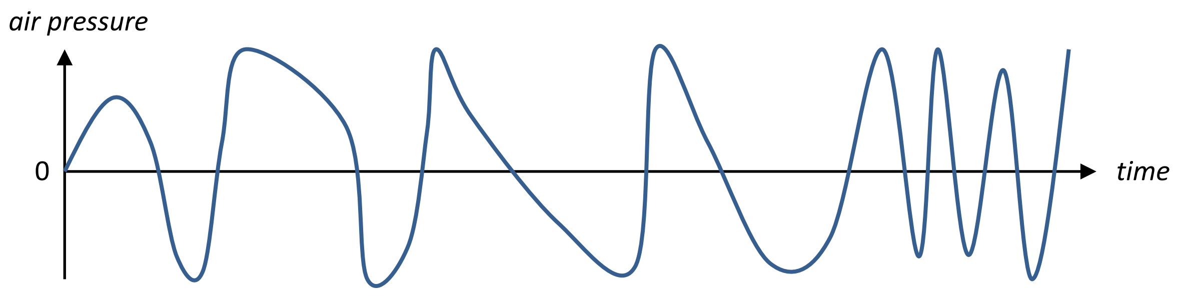
13 Digital Sound

A. What is sound?

It is a warm day. You pick up a magazine and start fanning yourself. You move the fan back and forth about 2 times a second, and you feel a nice breeze on your skin.

But suppose you start fanning yourself faster and faster. When you reach about 20 times a second, instead of feeling it on your *skin*, you will start to hear it in your *ears*. It will be a very low bass tone. The columns of air moving forward and backward make up "transverse waves" of air pressure, and this is picked up by your eardrums. As the air oscillates faster and faster, the pitch you hear rises. By the time it reaches 260 times a second, you hear a pitch that is roughly "middle C" on a piano.

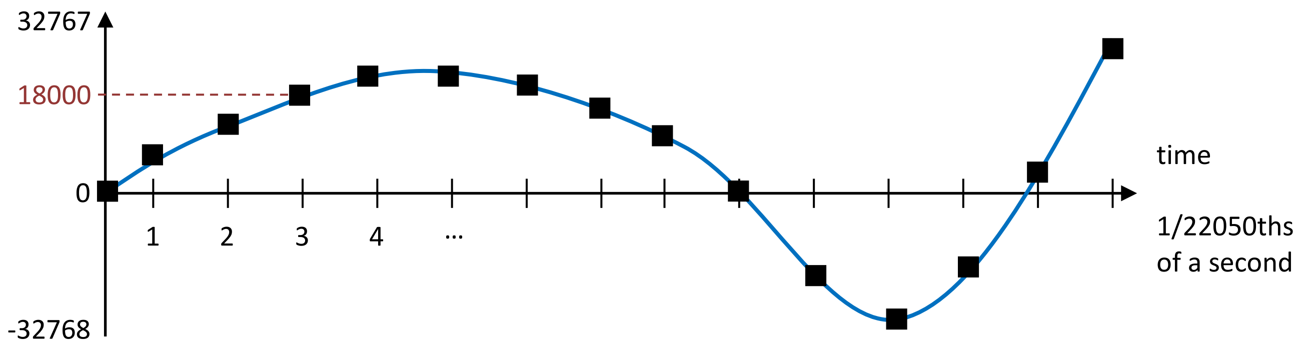
So sound is your ear's perception of air pressure pressing forward (positive) and backward (negative) through time. A loudspeaker pumps the air forward and backward to make sound. You could roughly imagine a graph like this:



This is an analog signal. Different shapes make different sounds. The vertical axis here, air pressure, is related to the loudness of the sounds. The bigger, the louder.

B. Going digital

To go digital, you need to chop up the horizontal and vertical axes into discrete chunks. To make this into a digital sound, we *sample* the sound signal repeatedly at regular time intervals. For example, we could sample it every 1/22050 of a second. The value of the sample will be an integer we call *intensity*, or sometimes (as JES calls it) simply the "sample value", ranging from -32,768 to +32,787.



The analog signal (blue curve) is replaced by just the separate black dots. Each black dot is called a *sample*. In fact, digital sound is made up of samples in the same way that digital images are made up of pixels. Samples in a digital sound are numbered 0,1,2,3… and so on. These numbers are called *indexes*. In the picture above, the sample value at index number 3 is 18000.

The number of samples per second is called the *sampling rate*. A digital sound with a higher sampling rate sounds more natural, in the same way that a digital photo with more pixels per square inch looks more natural. A sampling rate of 44,100 produces high quality digital sound, and is the standard for CD audio. A sampling rate of 8,000 is very low quality for music, but acceptable for voice (as in telephone conversations).

C. Accessing and manipulating digital sounds

JES provides a variety of functions for manipulating sounds. Digital sounds are stored in sound files, such **.wav** files. The most basic thing to do is load a sound from a file and play it:

```
file= pickAFile( )  # user picks a file such as a .wav or .au file
sound= makeSound( file )
play( sound )
printNow( "I hear music!" )
```

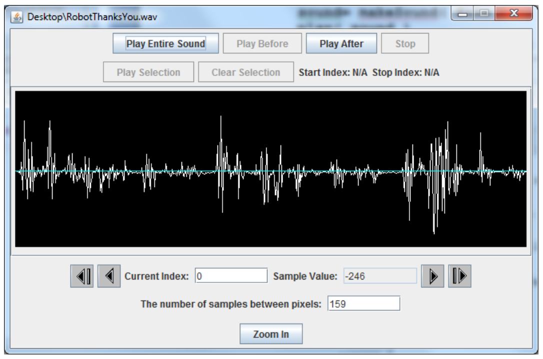
The **play** function will cause the sound to start playing. But after it starts the sound, it will *not* wait for the sound to finish. It will immediately go on to the next line of code. If you want the program to pause until the sound is finished playing, instead of **play** you should call **blockingPlay**.

Sounds are simpler that pictures in one way: they have one dimension (time) instead of two, and samples have just one number (intensity value), unlike pixels which have three (red, green, blue levels). But unlike pictures which you can stare at and study, sounds are fleeting. That's why it's nice to be able to visualize sounds by displaying their samples visually.

The **JES sound tool** does that. At the console you can open a sound from a file and call the **explore** function on it, and it opens an interactive tool for viewing:

```
sound= makeSound( pickAFile() )
explore( sound )
```

This will pop up:



It gives you a chance to "zoom in" on the sound, browse samples, and on. (If you want something a bit nicer, but still free, consider the Audacity tool at audacity.sourceforge.net. Audacity will allow you to convert your favorite mp3 files to formats like .wav so JES can open them with **makeSound**.)

Programs for processing digital sounds need to access sample values and change sample values. There are two simple functions for doing this. For example, consider this:

```
value= getSampleValueAt( sound, 3 )
```

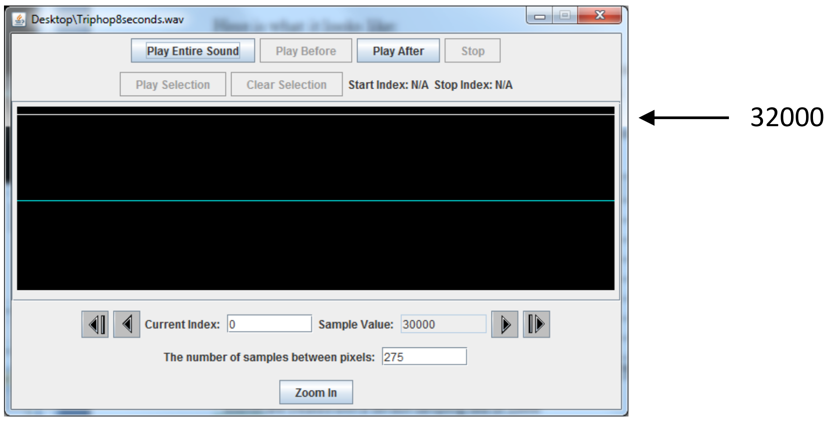
If the sound was represented by the figure on page 1, the value returned on the right would be 18000. If we wanted to change that value to -12500, we would make the call

```
setSampleValueAt( sound, 3, -12500 )
```

The following code uses the **getLength** function to get the number of samples that make up a sound, and then changes all the sample values in a loop:

```
length= getLength( sound )
for i in range( 0, length ):
  setSampleValueAt( sound, i, 32000 )
play( sound )
```

This sets all of the sample values to the same number: 32000. Here is what the resulting sound looks like:



What do you think this will sound like? Will it be one loud constant sound, since you set the intensity very high? No. It will be *silent*. The sound wave is a flat line. There are no air pressure changes, hence no sound.

Keeping the shape of a sound the same, but lowering the intensity values, will make a sound quieter:

```
sound= makeSound( pickAFile() )
length= getLength( sound )
for i in range( 0, length ):
  value= getSampleValueAt( sound, i )
  setSampleValueAt( sound, i, value/2 )
play( sound )
```

Just like we have **makeEmptyPicture** to make a blank picture of a given size, **makeEmptySound** will make a silent sound of a given length. The default sampling rate is 22050.

What happens if we make a sound consisting of just random sample values? Examine the following code:

```
sound= makeEmptySound( 50000 )
for i in range( 0, 50000 ):
  setSampleValueAt( sound, i, randrange( -32768, 32768 ) )
play( sound )
```

What will this sound like? It's called "white noise". Listen.

☐ Practice C

C1. What is the output of the following program?

```
def main():
   sound= makeEmptySound( 66150 )
   printNow( getDuration( sound ) )  # the duration of the sound (in seconds)
```

You should be able to figure out the answer before you run it, if you understand the meaning of sampling rate.

C2. What is wrong with the following code?

```
length= getLength( sound )
for i in range( 0, length ):
  value= getSampleValueAt( sound, i )
  setSampleValueAt( sound, i, value/2 )
  play( sound )
```

The following table summarizes the most important sound-related functions in JES.

Returns	Name	Parameters	Example
sound	`makeSound`	(string)	`snd= makeSound( filename )`
sound	`makeEmptySound` [1]	(integer)	`snd= makeEmptySound( numberOfSamples )`
sound	`duplicateSound`	(sound)	`snd2= duplicateSound( snd )`
none	`play`	(sound)	`play( snd )`
none	`blockingPlay`	(sound)	`blockingPlay( snd )`
integer	`getLength` [2]	(sound)	`length= getLength( snd )`
float	`getDuration`	(sound)	`seconds= getDuration( snd )`
float	`getSamplingRate`	(sound)	`rate= getSamplingRate( snd )`
integer	`getSampleValueAt`	(sound, integer)	`value= getSampleValueAt( snd, index )`
none	`setSampleValueAt`	(sound, integer, integer)	`setSampleValueAt( snd, index, value )`

[1] Sounds are created with a default sampling rate of 22050
[2] Same as **getNumSamples**

Learning Outcomes for Chapter 13

☐ Explain what sound is, and what it means for it to be digitally sampled.

☐ Be able to understand code that uses the sound functions in the table above.

☐ Be able to write short programs that use sound functions to achieve certain effects.

14 Simulation

A. Undecidable Games

Over the past three months you have explored *code*. As you know, code is text that tells a smart device what to do. Small programs are often called *scripts*, and the analogy to a drama script is very close. An actor follows a script; a computer follows a script too. Or, to put it more properly, the computer *executes* the *code*.

Drama scripts run in a straight path-- one line follows another. The actor starts at the beginning, and ends at the end. Certainly computer code can do the same:

```
fullname= firstname + " " + lastname
title= "Dr. " + fullname
salutation= "Dear " + title + ","
```

These three lines are executed one after another. But what gives code true power to make devices smart is that it allows *decisions* and *repetition*. These two notions come together in the **while** loop.

Here is a little program that uses a **while** loop in a simple way. Suppose someone gives you a pile of jellybeans. You split it in half and give half of them to a friend. (If you have an odd number, you are generous and give the friend the odd bean; for example if you have 9 beans you give the friend 5 and you keep 4).

```
beans= requestInteger( "How many jelly beans do you have?" )
printNow( "Number of beans to begin with: " + str(beans) )
count= 0
while beans > 1:
  # Give away half your beans (rounding down if an odd number)
  beans= beans / 2
  printNow( "Beans left: " + str(beans) )
  count= count + 1
printNow( "Friends who got beans: " + str(count) )
```

Suppose you start with 100 beans. You run this program and get the following output:

```
Number of beans to begin with: 100
Beans left: 50
Beans left: 25
Beans left: 12
Beans left: 6
Beans left: 3
Beans left: 1
Friends who got beans: 6
```

This says if you start with 100, you can cut it in half 6 times before you get down to 1. Suppose you start with 1000 beans? Running the program shows you can cut it in half 9 times before you get down to 1. If you have a million beans, you can cut it in half 19 times. This number is called the *logarithm to the base 2*. It is the number of times you can cut something in half before reaching 1.

As you know, the problem with **while** loops is that they sometimes never stop. For example, suppose you change the **while** loop in the code above to this:

```
while beans >= 0:
  # Give away half your beans (rounding down if an odd number)
  beans= beans / 2
  printNow( "Beans left: " + str(beans) )
  count= count + 1
```

The code will never stop: beans will always be greater than or equal to 0. This is an infinite loop.

Now let's try another variation on the jelly bean scenario:

> *You start with a pile of jelly beans. If the number is even, you give away half of them. If the number is odd, someone gives you twice as many jelly beans as you have already (so now you have 3 times what you had before), and tosses in an extra one too. All this stops when you are down to 1 bean.*

Think about this. It looks a little... unbalanced. True, you sometimes cut your supply of beans in half, but you might also more than triple your supply. It seems like you might go on forever getting lots and lots of beans.

Let's express this in code to see what happens:

```
beans= requestInteger( "How many jelly beans do you have?" )
printNow( "Number of beans to begin with: " + str(beans) )
while beans > 1:
  if beans % 2 == 0: # beans is even (remainder when divided by 2 is zero)
    beans= beans / 2
  else:
    beans= 3*beans + 1
  printNow( "Beans left: " + str(beans) )
```

Suppose we start with 11 jellybeans. Here's what the program tell us will happen:

```
Number of beans to begin with: 11
Beans left: 34
Beans left: 17
Beans left: 52
Beans left: 26
Beans left: 13
Beans left: 40
Beans left: 20
Beans left: 10
Beans left: 5
Beans left: 16
Beans left: 8
Beans left: 4
Beans left: 2
Beans left: 1
```

Your supply of jelly beans goes up and down, but eventually you are down to 1 bean. We suspect this ritual can get kind of long. So instead of printing one line of output, let's just pile all the successive beans counts into a list, and print the list out at the end.

```
beans= requestInteger( "How many jelly beans do you have?" )
list= [beans]
while beans > 1:
  if beans % 2 == 0:
    beans= beans / 2
  else:
    beans= 3*beans + 1
  list= list + [beans]
printNow( list )
```

When we run this version, we get the more concise output:

```
[11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]
```

Now suppose we start with 27 jellybeans. Running the program tells us that for a while we will be in possession of a *lot* of beans, but then they all go away:

```
[27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121,
364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526,
263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754,
377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079,
3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154,
3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8,
4, 2, 1]
```

Notice how we went up to 9232 beans at one point! But, starting from a few more jellybeans, 32, leads to a very quick game:

```
[32, 16, 8, 4, 2, 1]
```

This kind of program is called a **simulation**. In a simulation, we take something that could happen in the world (like a person playing a little game with beans) and replicate it in the computer. With the computer, we run this whole game billions of times faster than in the real world, and this allows us to explore different possibilities. We can simulate simple games, we can simulate economies, ecosystems, solar systems-- you name it. It is one of the chief applications of computer science.

If a system is simple, you don't really need a computer to simulate it. You can just "do the math". That's the case of first game in this chapter, when we simply cut the beans in half. The number of times through the while loop is given by something every math major will learn: the logarithm to the base two.

Our second jellybean game was only slightly more complex, but that tiny extra complexity is all it takes to make "doing the math" impossible. We *must* do a computer simulation! Is there a formula to tell us how many times the while loop repeats? No. In fact, we can even ask a more basic question:

> **Question:** Could this ever be an infinite loop? That is, could we start with a certain number of jellybeans and find that, following this game, the number goes up and down and up and down, but overall with more "up" than "down", so that the number never diminishes to 1?

Think about it for a minute.

This is a very, very difficult question. *In fact, to this day, no one knows the answer!* The conjecture that this is *never* an infinite loop, no matter how many beans you start with, is known as the **Collatz Conjecture**[9]. But no one has been able to prove this. For all we know, there may be a number of beans that yields a never-ending game.

So: deciding whether a piece of code has an infinite loop can be very difficult indeed.

In fact, it's worse than that.

You can actually change the rules of the jellybean game in a way that you can *prove that it is unprovable* whether the **while** loop is infinite!

[9] Feel free to look this up, on Wikipedia for example.

In fact, computer science is filled with proofs that certain questions about code are unanswerable. The most important result in the theory of computing is **Rice's Theorem**, which says, roughly, any interesting question about what a piece of code does is unanswerable. This includes asking about a piece of code, "Is it malicious?"

This is a profound result, and is one of the deepest results in science. The very features that make code powerful (*decisions* and *repetitions*) are precisely what make questions about code unanswerable.

B. Stochastic Simulation

The little jellybean game in the previous section had one special property. If you start with the same number of jellybeans, you always get the same result. The way you started out completely determined what happened. Simulations like this are called **deterministic**.

Other simulations, by contrast, can involve randomness. For example, if you roll some dice as part of a game, starting the same way twice can give two different outcomes, depending on how the dice come out. When we simulate such situations with computer programs, we can do this by asking the computer to give us random numbers. (Your instructor may have shown you the Birthday Surprise simulation, which does this.) Such simulations that involve randomness are called **stochastic**.

Here is problem that involves randomness that we can tackle with a stochastic simulation. (It is related to the "drunk in the minefield" example that you may have done in class or in a the lab.)

Suppose you start at the center of a 200x200 fenced-in field. You start wandering by taking random steps. A random step means that you pick a horizontal move randomly (one unit left, stationary, or one unit right) and a vertical move randomly (one unit up, stationary, or one unit down). **Question**: On average, how many steps will it take before you hit the fence?

We can start by writing a function that simulates a single random walk from the center of the field to the edge.

```
def walkToFence( w, h ):
# Return how many steps it takes to randomly walk from the center
# of a w-by-h field to its edge.
  x= w/2
  y= h/2
  steps= 0
  while x>=0 and x < w and y >= 0 and y < h:
    x= x + randrange(-1,2)
    y= y + randrange(-1,2)
    steps= steps + 1
  return steps
```

We can repeatedly run this function and compute the average number of steps:

```
def simulate( numReps, s ):
# Report on the average number of steps it takes to randomly walk from
# the center of an s-by-s field to its edge, given a number of repetitions.
  totalSteps= 0
  for reps in range(0, numReps):
    totalSteps= totalSteps + walkToFence( s, s )
  averageSteps= totalSteps / numReps
  printNow( "Average number of steps from center to edge: " + str( averageSteps ) )
```

At the console, we can run various simulations. Let's try to answer our question (about a 200x200 field) by averaging the results of 4 runs:

```
>>> simulate(4,200)
Average number of steps from center to edge: 7935
```

Let's try that again:

```
>>> simulate(4,200)
Average number of steps from center to edge: 3907
```

Big difference. There seems to be a lot of uncertainty here. Let's take the average over 100 runs.

```
>>> simulate(100,200)
Average number of steps from center to edge: 8867
```

And again:

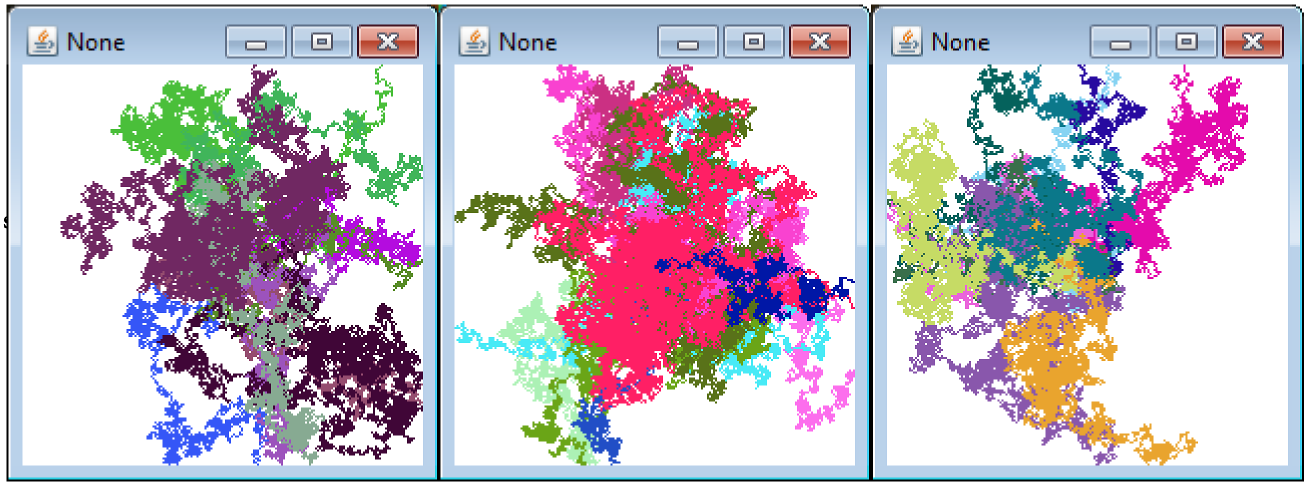
```
>>> simulate(100,200)
Average number of steps from center to edge: 9138
```

By averaging over larger number of runs, we seem to get less variation. This is a key part of stochastic simulation: we use large numbers of runs to get more certainty. Exactly how many runs we need to be confident of a result is something that scientists study carefully. This requires a solid knowledge of statistics.

Another, less formal, way to give us confidence in the correctness of a simulation is to use **visualization**. For example, it might be insightful if we could actually draw the random walks on a picture, to get a sense of their shape. Since we are doing multiple random walks, we could draw the path of each walk in a different random color. Here is a variation of this program that adds visualization:

```
def walkToFenceV( pic ):
# Return how many steps it takes to randomly walk from the center
# of a picture to its edge.
  w= getWidth( pic )
  h= getHeight( pic )
  color= makeColor( randrange(0,256), randrange(0,256), randrange(0,256) )
  x= w/2
  y= h/2
  steps= 0
  while x>=0 and x < w and y >= 0 and y < h:
    setColor( getPixelAt(pic,x,y), color )
    x= x + randrange(-1,2)
    y= y + randrange(-1,2)
    steps= steps + 1
  return steps

def simulateV( numReps, s ):
# Report on the average number of steps it takes to randomly walk from
# the center of an s-by-s field to its edge, given a number of repetitions.
  totalSteps= 0
  pic= makeEmptyPicture( s, s )
  show( pic )
  for reps in range(0, numReps):
    totalSteps= totalSteps + walkToFenceV( pic)
    repaint( pic )
  averageSteps= totalSteps / numReps
  printNow( "Average number of steps from center to edge: " +
str( averageSteps ) )
```

Here is the output from three calls to **simulateV(10, 200)** :



This shows that there is a slow random drift to the edge, and gives us a sense that many positions are revisited multiple times.

In sum, in this chapter we have had the chance to mention a key result from *within* computer science (Rice's Theorem), and gave illustrations of how computer science can contribute to the advancement of *other* sciences through simulation.

Learning Outcomes for Chapter 14

☐ Explain the difference between deterministic and stochastic simulation.

☐ Give an example of a simple deterministic simulation with results that are seemingly unpredictable.

☐ Explain the importance of repetition (multiple runs) and visualization in stochastic simulation.

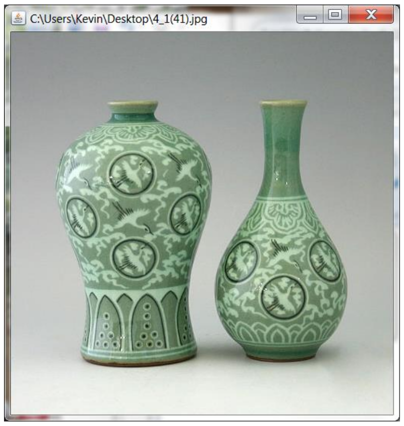
Appendix A: Code Example Sampler

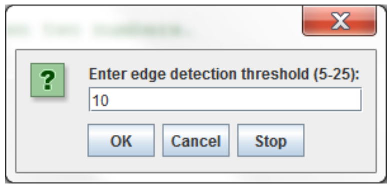
1. Piano Jazz

```
# PianoJazz.py
# INF 120 Fall 2009
# ------------------------------------------------------------------------------
# Shows the use of random numbers and the MIDI playNote function.
# ------------------------------------------------------------------------------

from random import *

def main():
  # The midi note to base the random melody on.
  tonic= 60  # middle C

  # The duration (in milliseconds) of the shortest note.
  tempo = 100

  # Favor steps that are in the major scale.
  stepDistribution= [0,0,0,1,2,2,2,3,4,4,4,4,5,5,5,5,6,7,7,8,9,9,10,11,11,12,12]

  # Make durations powers of 2.
  durationDistribution = [1,1,1,1,2,2,3,4]

  # Play 72 notes.
  for noteCount in range(1,72):
    step= choice( stepDistribution )
    duration= choice( durationDistribution )
    loudness= randrange( 100,128 )
    playNote( tonic+step, tempo*duration, loudness )
    playNote( tonic, tempo*duration, loudness - 10 )

    # Change the tempo up/down every 32 notes.
    if noteCount % 64 == 0:
      tempo *= 2
    elif noteCount % 64 == 32:
      tempo /= 2

    # Change the tonic every 16 notes.
    if noteCount % 16 == 0:
      rand= randrange(0,4)
      if rand==0:
        tonic += 5
      elif rand==1:
        tonic -= 5
      elif rand==2:
        tonic -= 12
      else:
        tonic += 12
      # Ensure we don't fall off the end of the keyboard.
      if tonic < 12:
        tonic += 36
      elif tonic > 110:
        tonic -= 36

  #End nicely on the tonic.
  playNote( tonic, 4000, 127 )
```

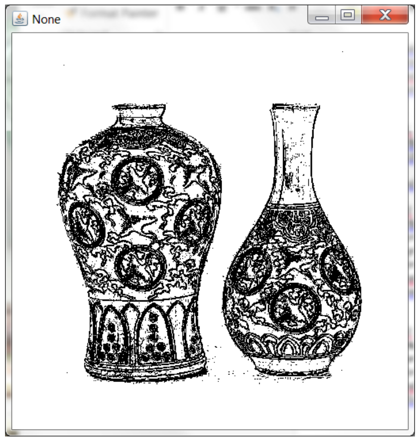
2. Edge Detector

```
# EdgeDetect.py
# INF 120 Spring 2009
# ----------------------------------------------------------------------
# Outlines the edges in an image.
#
# The threshold indicates how big the difference in intensity between
# neighboring pixels should be to count a pixel as an edge pixel.
# The lower the threshold, the more "edges" in the image.
# ----------------------------------------------------------------------

def main():
  pic= makePicture( pickAFile() )
  show( pic )
  threshold= requestInteger( "Enter edge detection threshold (5-25): " )
  edgePic= makeOutline( pic, threshold )
  show( edgePic )


def difference( a, b ):
# The positive difference between two numbers.
  if a > b :
    return a - b
  else:

def intensity( px ) :
# The intensity of a pixel's color (higher means lighter).
  r= getRed( px )
  g= getBlue( px )
  b= getGreen( px )
  avg= ( r + g + b ) / 3
  return avg


def makeOutline( pic, threshold ):
# Make an image consisting of the edges in pic.
  w= getWidth( pic )
  h= getHeight( pic )
  edgePic= makeEmptyPicture( w, h )
  for x in range(1,w) :
    for y in range(1,h):
      px= getPixel( pic, x, y )
      pxLeft= getPixel( pic, x-1, y )
      pxUp= getPixel( pic, x-1, y-1 )
      leftDiff= difference( intensity(pxLeft), intensity(px) )
      upDiff= difference( intensity(pxUp), intensity(px) )
      if leftDiff > threshold or upDiff > threshold :
        # px is an edge pixel!
        setColor( getPixel(edgePic,x,y), black )
  return edgePic
```




Enter edge detection threshold (5-25):

10



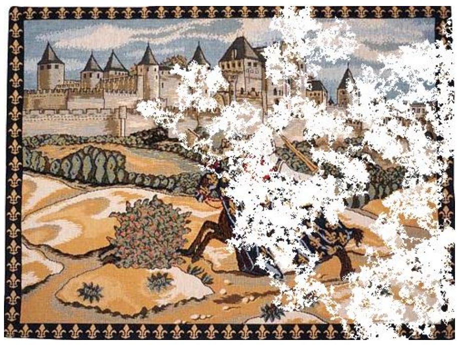
Appendix B: Programming Assignment Sampler

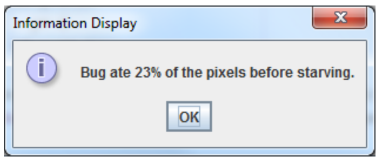
nku inf 120 spring 2011 | jython programming project 3

The purpose of this assignment is to give you practice in **while loops, function calls** and **if** statements. Your program will do a little animation/simulation of a bug eating up an expensive tapestry.

The idea is that a hungry little bug starts at the center of a valuable tapestry, and starts munching away. It "eats" each pixel, turning it to white. It moves randomly up/down/left/right, just like in the **Escape.py** example we did in class. As it wanders around eating, it leaves a trail of damage.

It does not go outside the picture. It continues wandering and eating until it has encountered 400 white pixels in a row, meaning it has eaten through so much of the tapestry it hasn't run into anything remaining to eat. The program then prints out how many pixels the bug ate. Here is a before-and-after picture from one example:





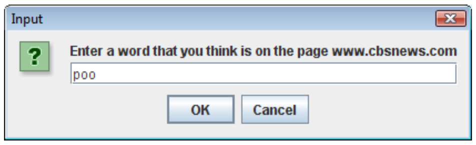
Bug ate 23% of the pixels before starving.

In pseudocode:

```
Have the user pick a picture from a file, and show it.
Get its width (w) and height (h).
Assign variables x and y to the center of the picture.
Let pixelsEaten be the number of pixels the bug has eaten; initially set it to 0.
Let hungerLevel count the number of consecutive steps the bug has eaten nothing; initially set it to 0.

Repeat the following until hungerLevel reaches 400:
        Get the pixel at position (x,y) in the picture
        If the color is white
                The bug has encountered a blank spot!
                add 1 to hungerLevel
        else
                The bug is back on the tapestry and "eats" the pixel !
                set the pixel to white
                add 1 to pixelsEaten
                set hungerLevel back to zero
        Repaint the picture
        The bug moves (up to) one pixel horizontally: add a random number (-1,0,or1) to x
        The bug moves (up to) one pixel vertically: add a random number (-1,0,or1) to y
        If x goes off the picture to the left, reset it to 0
        If x goes off the picture to the right, reset it to w-1
        Do a similar thing for y if it goes off the top or bottom
In an information box display a message showing the percentage of the tapestry that the bug ate.
```

For this assignment you will write two little functions (less than a dozen lines each) that give you a chance to practice looking up technical information to help you solve problems.

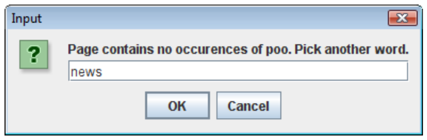
1.
The first function will be called **webwords**. It is a little game that asks the user to guess a word that appears on a website (without seeing the website). For example, if I type `webwords("www.cbsnews.com")` at the console, it will prompt me to enter a word in a text box that I think occurs on that page's source. Suppose I enter this:


Input

Enter a word that you think is on the page www.cbsnews.com

poo

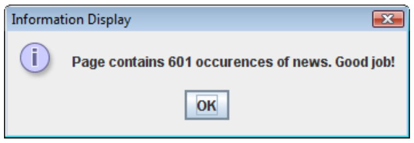
Since my word does not occur on that page, it asks me again. This time I try a more reasonable word:


Input

Page contains no occurences of poo. Pick another word.

news

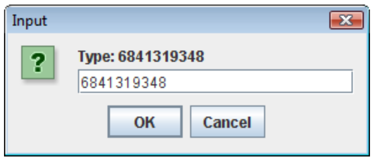
and in this case, my guess was good. It pops up:


Information Display

Page contains 601 occurences of news. Good job!

and the function ends.

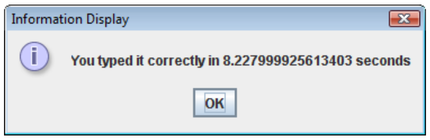
2.
The second function will be called **typingTimer**. It times how long it takes me to type a random string of digits. If I enter `typingTimer( 10 )` at the console, it picks a random 10 digit string and asks me to type it in:


Input

Type: 6841319348

6841319348

When I hit enter (or OK) it pops up how well I did:


Information Display

You typed it correctly in 8.227999925613403 seconds

If I type it wrong, it just says "Incorrect".

Design hint: Repeatedly pick random numbers 0–9 and convert them to strings, repeatedly appending them to an initially empty string (`""`) with `+` until it's long enough.

Seeking out technical advice on available functions is always a big part of writing scripts and programs. The following "Knowledge Base" will tell you everything you need to know for this assignment.

Q. How do I access a web page in Python?

A. At the top of your file, put the line

```
import urllib
```

The following code stores the source of the entire web page at **www.nku.edu** into one very long string named **page**:

```
connection= urllib.urlopen( "http://www.nku.edu" )
page= connection.read()
connection.close()
```

Don't forget the http://. Of course, you have to be connected to the Internet.

(If you are unfamiliar with the "source" of a web page, go to any web page in your browser and pull down *View* menu and select *Page Source* (on Firefox), or the *Page* menu and select *View Source* (on Internet Explorer), to see what it looks like. You will learn more about this in INF 286.)

Q. How do I count the number of times a certain string occurs in another string?

A. Study the following example:

```
s= "the end of the world as we know it"
print s.count( "the" )    # prints 2
print s.count( "dog" )    # prints 0
print s.count( "e" )      # prints 4
```

(The odd syntax of count is new. It is a lot like a function, but because of the dot it is called a "method" on the "object" **s**. You will learn more about objects and methods in future courses.)

Q. How do I time how long something takes?

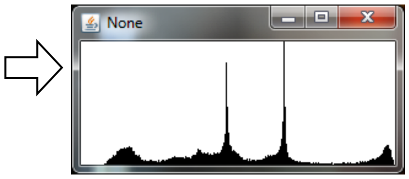
A. At the top of your file put the line

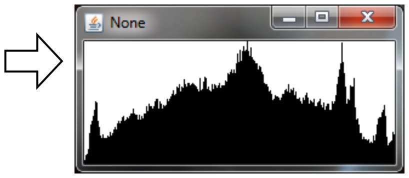
```
import time
```

The function `clock()` returns the number of seconds since JES started. If you call this function twice at two different points in your program and subtract the last from the first, it will give you the elapsed time.

nku inf 120 spring 2011 | jython programming project 5

For this assignment you will write a program that gets a picture from the user and produces an **image histogram** for it. An image histogram is a way of representing how many pixels are present at each brightness level. Remember *brightness* is the average of the red, green, and blue levels, and ranges from 0 to 255. Lower numbers mean darker, higher numbers mean lighter. Here are some examples:





In the histograms on the right, the *x* axis (going rightward) represents increasing brightness from 0 to 255. The height of the vertical line of black pixels represents the relative percentage of the pixels at that level of brightness.

How do you do this? The idea is to first create a list named **pixelCount**, with 256 slots. Initially it will be filled with zeros. For each pixel in the image, your program will compute its brightness **i** (the average of its R,G,B values). It will add 1 to the running total **pixelCount[i]**, which is the number of pixels in the image with brightness **i**. At this point, be sure sure to review the **LightMedDark.py** example we did in class on March 15!

Once your program has finished calculating the **pixelCount** list, it will compute the maximum value in that list. To do this, use a loop that keeps a running maximum. The relative percentage of pixels with brightness level **i** will be **100* pixelCount[i]** divided by this maximum.

Finally, you will display the histogram. You will create an empty 256x100 picture, and loop for **x** from 0 through 256. You will use the **addLine** function to draw the vertical black lines. The length of the line at horizontal position **x** is the relative percentage of pixels with brightness level **x**, as calculated above.

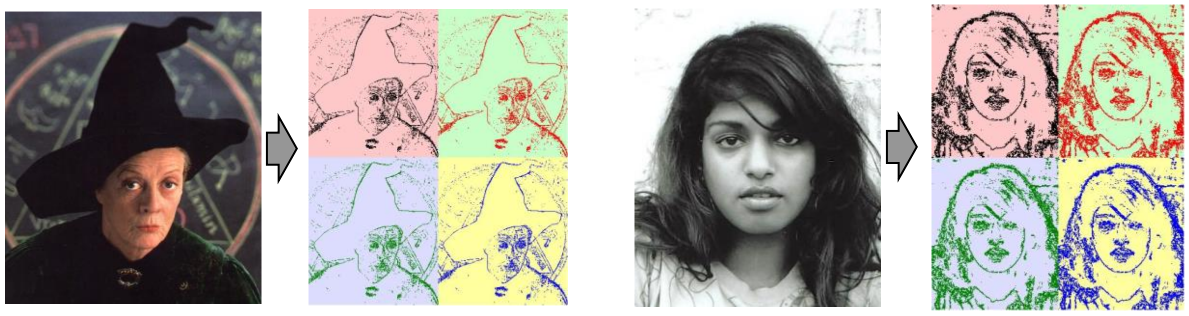
Your program will ask the user for a picture, calculate the histogram, and then display it. Polish your code: make sure comments, whitespace and variable names are all chosen wisely. Be sure to test it. (Try it out on solid color pictures, for example. What would you expect to see?)

nku inf 120 / jython programming assignment 6

Here is the capstone to our image programming adventure. If you do an image search on the web for "**Warhol Marilyn**". You'll find images like the one in the corner above.

For this program, you will write a "warholizer" that will take any picture and produce a new picture of the same size consisting of a 2x2 grid of outlined copies of the images with different dark color edges and different light color background.

Instead of Marilyn Monroe, here's Minerva McGonagall and MIA:



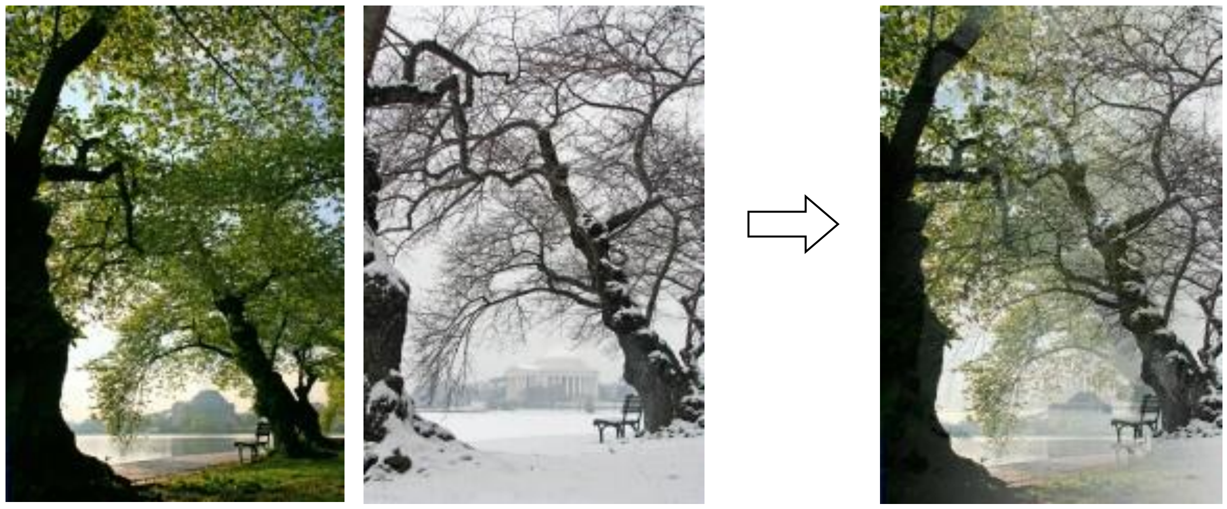
Your program will use the edge detection function we developed in class (now posted on Bb). You can use that function as-is. Here's some pseudocode.

```
def warholize( pic ):
# Returns a 2x2 warholized version of an image
    Let w,h be its width and height.
    Let picEdge be the "edge outline" version of pic, using the function we wrote in class.
    Let picNew be a new empty white picture of the same size.
    for x in range(1,w,2):
        for y in range(1,h,2) :
            let c be the color of the pixel in picEdge at location x,y (it will be black or white)
            # Upper left quad
            let px be the pixel in picNew at (x/2, y/2)
            if c is black (meaning it is an edge pixel)
                set the color of px to a dark color (your choice)
            else:
                set the color of px to a light color (your choice)
            # Upper right quad
            let px be the pixel in picNew at  (w/2+x/2,  y/2)
            if c is black:
                set the color of px to a dark color (your choice, but different from above)
            else:
                set the color of px to a light color (your choice, but different from above)
            # Similarly for other two quads …
    return picNew
```

Read the pseudocode first and make sure you understand it before typing in any Python. Why are we only looking at even-numbered pixels of the edge image? Because each quad is half the width and half the height, we only need refer to half the number of pixels in each direction.

Have your **main** function ask the user to pick a file. It should show the original, then show the resulting "warholized" picture. The threshold for detecting edges will be defined in your code; do not bother the user by asking for this value. Pick a value that makes the result look decent.

For this assignment, you take two pictures of the same height and width. When you look at the result, you see on the left side the first picture, but it gradually fades into the right picture.



Let's think of the algorithm. You will construct a new image from the two originals (see **duplicatePicture** in JES). The RGB values of each pixel in the new image are weighted averages of the RGB values of the pixels in the originals. For example, when you start out on the left side, the colors are 100% from the first image and 0% from the second image. As you move rightward the proportion will change gradually, so that when you get about a quarter of the way across it will consist of 75% from the first image and 25% from the second image. In the middle of the image it will be 50%-50%. And so on.

Generally: As you move across a row, **x** increases from **0** to **w**–1 (where **w** is the width of the image), and the amount of the *second* image that gets mixed into the first would be given by:

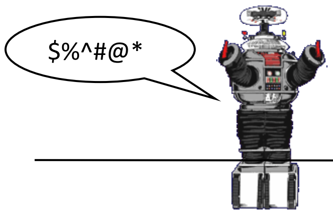
mixingLevel= x / float(w-1)   ***# Do you see why we need* float*?***

This a number between 0 and 1. Then **1 – mixingLevel** will tell you how much comes from the *first* image pixel. So for example if **r1** and **r2** are the red levels from the two images, then the red level at the corresponding pixel in the new image will be **int(mixingLevel * r2 + (1–mixingLevel) * r1)**. Notice we need to convert these floating point numbers back to integers using the **int** function. Try it out on scratch paper with a tiny (5x5) image to make sure this makes sense.

For this program you will write a program that consists of the following functions:

crossfade: Takes two pictures as parameters, and returns a new picture as described above.

main: Gets two pictures from the user. If the second picture is not the same size as the first, the program pops up a message box that tells the user the width and height of the two images and points out they should have been the same size, then ends. Otherwise it displays the result of cross-fading the image.

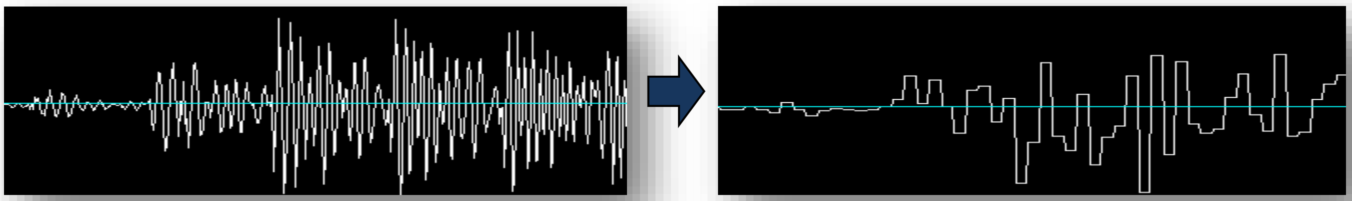


We have seen how digital sound is represented by samples at a given sampling rate. If the sampling rate We have seen how digital sound is represented by samples at a given sampling rate. If the sampling rate becomes very low, the sound becomes highly distorted. **This assignment asks you to write a little game in which the user tries to recognize human speech when the audio is highly distorted.**

The type of distortion we will use is called **flattening**. When distorting a sequence of numbers to level *L*, each contiguous run of *L* numbers is the same as its first number. For example:

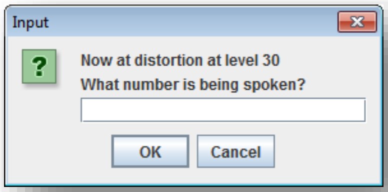
original sequence: [564, 234, 100, 432, 800, 923, 753, 291, 333, 811, 443, 222, 702, 211]
distorting to level 2: [564, 564, 100, 100, 800, 800, 753, 753, 333, 333, 443, 443, 702, 702]
distorting to level 4: [564, 564, 564, 564, 800, 800, 800, 800, 333, 333, 333, 333, 702, 702]

When applied to the sample values in a digital sound, you can see this would make the waveform consist of a sequence of flat areas (and the flat areas get wider as the distortion level increases):



On the Blackboard site there is a set of sound files with names **MysterySpeech*.wav**. These are approximately two seconds in duration, and consist of a human pronouncing a number between 10 and 99. The number spoken is stored as the value of the first sample. For example, if the file contains someone speaking "eighty-two", then the sample value at position 0 in the sound file is 82. (This is inaudible[10].)

Here is the user's experience of your program. It tells the user to pick one of the "Mystery Speech files". It distorts the sound to level 30, plays it, and asks the user to guess the number spoken. If the guess is correct, the program pops up a message box saying "Correct." Otherwise it reduces the distortion to 25 and lets the user try again. It keeps reducing the distortion level by 5 until it drops to 0, then the game is over. At the end of the game, it plays the undistorted sound. The program then asks the user if s/he would like to play another game. If so, it does, repeating the above; if not, the program terminates.


Input

Now at distortion at level 30
What number is being spoken?

Make sure you break up this program into several functions. For example, your **main** might call a function that plays one game. Another useful function would be one that takes two parameters - a sound and a distortion level - and returns the result of distorting that sound.

Start early, test as you go. Your code must be clear: good names of functions and variables, helpful commenting, and clear layout are essential.

Concepts reinforced: *lists, digitizing signals, loops, decisions, behavioral decomposition, user interface.*

[10] This is an example of what is called "steganography", in which secret messages are stored in pictures or sound.

lookitup!

spring 2008 nku inf 120 / jython programming assignment 7 :synaesthesia

This program is about listening to colors. We will treat pictures as piano scores.

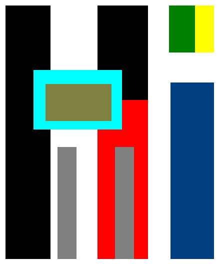
Write a function called **listenToPicture** that takes one picture as an argument. It first shows the picture.

Next, it will loop through every 4th pixel in every 4th row[11] and do the following. It will compute the total of the red, green and blue levels of the pixel, divide that by 9, then add the result to 24. That number will be the note number played by **playNote**.

That means that the darker the pixel, the lower the note; the lighter the pixel, the higher the note. It will play that note at full volume (127) for a tenth of a second (100 milliseconds). Every time it moves to a new row, it prints out the row number (y value) on the console.

Your main function will ask the user to select a file with a picture. It will print the number of notes to be played (which is the number of pixels in the picture divided by 16; why?). It will then call the **listenToPicture** function.

Try it on the following three pictures available on the Blackboard site. The first will allow you to easily see/hear if your code is working correctly.


simplePic.bmp

miaTiny.jpg

johnlennonTiny.jpg

Come up with your own small image that sounds beautiful, either by drawing it in Paint (and saving it as a .bmp file) or finding a jpg that is pretty.

[11] That means in row y=1 it will get the pixels at x=1,5,9,13,17, … until the end of the row. Then it will move on to row y=5 and do the same, then y=9, etc. This is clearly a loop within a loop.

nku inf 120 fall 2010 | jython programming project 8

This program is a variation on the “Birthday Surprise” simulation we did in class (and which is also on the slide set). In that case, we wrote a program to answer the question: *In a class of a given size, what is the chance that there are students born on the same day of the year?*

Here is the research question for this project: *In a class of a given size, what is the chance that there are students born in each of the twelve months of the year?* That is, we are looking for the following case: for every month Jan-Dec there is at least one student in the class whose birthday is in that month. Obviously you need at least 12 students in the class for this to happen.

One could study this using the mathematics of probability, but, as with the Birthday Surprise, we will tackle this using computer simulation.

Your program requires no input from the user. It will simply print the following table on the console, for all class sizes from 20 to 60:

```
CLASS SIZE   CHANCE OF BIRTHDAYS IN ALL 12 MONTHS
20           5%
21           7%
22           10%
… (etc.) …
59           92%
60           93%
```

(Your results may vary… a bit. Why?)

To compute a decent estimate of the percentages, for each class size have your code run 10000 simulations, then take the number of those simulations that produce birthdays in all months, and divide by 100.

As we did with birth *days* in class, you may assume that birth *months* are assigned “randomly and uniformly”. This means that you can use **randrange** to pick a number between 0 and 11 to assign a birth month to each student in a class.

You should decompose what needs to be done into simple, general functions. Your code must be clear: good names of functions and variables, good choices of parameters, helpful commenting, and clear layout are essential. In particular, be sure to comment the beginning of each function explaining what it does.

As always, start early, test as you go.

Concepts reinforced: *lists, loops, running totals, decisions, behavioral decomposition, simulation*