START HERE:

Python 3x Programming

Made Fun and Easier

by
Jody Scott Ginther

This Free sample of the course gives you the first Five lessons to see if you are interested in the full course and introduce you to Python programming. Order the full course at www.tooncat.com or www.Quantum-Sight.com

Cartoons by Jody S. Ginther www.toonzcat.com

Start Here:

Python Programming Made Fun and Easier

by

Jody Scott Ginther

Free Version

Published by
Alien Cat Studios ®

Quantum-Sight ™
(Educational Division)
1712 W Stevens #D
Bozeman, MT 59715

THE CONTROL NUMBERS FOR THE FULL E-BOOK COURSE/VERSION ARE:
ISBN 9780-98938471-1

THE CONTROL NUMBERS FOR THE PAPERBACK VERSION ARE:

ISBN 9780989384704

Library of Congress Control Number: 2013907954

Printed in the United States of America
First Printing May 2013

DEDICATION

To my two sons, who helped me understand the deeper meaning of:

"Never give up!"

The next page is intentionally left blank just to bother people.

You will never know...

CONTENTS

ACKNOWLEDGMENTS

It is not possible to acknowledge by name everyone who has influenced me in writing this course. So, I will have to offer a generic "thank you" and acknowledge those unknown people who influence me along the way. I would like to acknowledge the people at Python.org for their unwavering support for Python programmers. I would also like to acknowledge those throughout the Python community who may have had a part in providing or influencing some of the code used in this course.

The next page is intentionally left blank just to bother people.
You will always wonder.

Introduction

Who Should Read This Book?

- Anyone who wants to learn programming in Python
- Those souls mastering the art of independent learning
- Programming Teachers with a sense of humor
- People who are curious
- People who are bored and have nothing better to do
- You

The unexpected popularity of my mini-book on Python programming combined with constant requests to write a more in depth course; inspired me to write this one. The first book, “Start Here: Python Programming Made Simple for the Beginner” was a free introduction to Python programming. I had no idea that over 10,000 copies of this short e-book would spread worldwide in the first two months from one unknown website. This opened my eyes to the need for programming books and courses that teach Python in an interesting or at least “different” way.

This book includes an updated, and re-written course that includes all the contents of the first book. So, it is unnecessary to read that book first. You can begin here. Content changes and improvements have been made to each chapter and we go deeper into the world of Python in this course. I use the term “course” rather than “book” because I have added some self-study exams, exercises, and quizzes. The intent is to use this as a programming course for classrooms; and a resource for those who prefer independent-study. When you finish this book, you will be able to do some programming in Python, make simple games, package them, and be well down the road to being a great programmer.

Why isn’t this course free like the first mini-book? My wife likes to eat, so I guess I should feed her once in awhile. Unfortunately my time for free projects is limited.

The author, (that’s me if you haven’t guessed already), believes the theory that visual learning, humor, (emotional learning), interactive or conversational communication, and action, (experiential learning), are some of the best ways for most people to quickly learn something. I attempt to be as brief as possible to transform the reader into a competent programmer as fast as possible. This means I did skip things about history, comparisons to other programming languages you probably don’t know anyway, and anything else that may slow you down, (other than humor and visual elements to help you remember things and reduce boredom; yours and mine).

Sincerely Mine,

Jody S. Ginther

Who's The Author?

WHO CARES? "***I just want to get into programming quickly!***" <u>*Then skip this part*</u>. The author, (Jody S. Ginther), chose to include only a few useless details about himself that you probably don't care about anyway, *(yeah it's a man with a girl's name, kind of like "Jackie" Chan).* If you must know a little about him; he has taught internationally at major universities for over 17 years, (Jody not Jackie Chan). He has a wide range of interests and has taught computers, programming, math, business, science, languages, and martial arts. He has and continually researches effective learning methods to speed up the learning process. His hobbies include; programming, learning new things, directing, writing, art, playing drums, guitar, piano, singing, making movies, cartoons, and acting. Anyway, enough meaningless drivel about that guy, the focus in this course is not on useless information about that one, but is all about:

turning you into a programmer as quickly as possible.

(Empty white space for no apparent reason)

How To Use This Course

Even if you read my previous short e-book; “Start Here: Python Programming Made Simple” you should still begin at Lesson 1 in this course. One reason is because it is always good to reinforce and review even if you already read most of this information. But most importantly, new information and expanded explanations have been added that may be important to you.

If you are an experienced programmer jump around at will to the parts you don’t know; need to review; use this as a reference; or just read everything anyway for its entertainment value.

If you are new to programming this is the course to begin with. Although the lessons in this course have been made simple to understand; keep in mind that intermediate and advanced programming is not so simple when you are new to it. But, just as any skills or talents, anything you practice enough will become easier and more simple as you become a master. A **master** *is someone who has done some skill so many times; it looks easy and has become easy for them to perform and teach.*

Don’t just read the book unless you plan to read a lesson one time through and go back to do it the second time through. Just as speaking any language can only truly be learned by speaking it; learning a programming language can only be mastered by doing it. Each code example is intended for you to do, not just to read. You will be surprised that things you thought looked too simple to bother typing, need to be practiced. The elimination of a space or a mild typo will destroy your program. You must practice as much as possible until you are a code perfectionist. This will save you countless hours of debugging or fixing your code and will radically lower your level of stress.

In this course, self-testing proficiency exams and quizzes are provided to help you understand how well you know the information that has been covered to that point. Depending on your style of learning, **you could approach these lessons in different ways**:

1. **The best way for most people is to do each lesson, type, (don’t copy and paste), all code as you follow along. *Why?* To re-enforce what you learn by doing it, and because the most common errors in programming come from typing and copying mistakes. If you don’t completely understand the lesson, repeat it until you do before going on.**
2. **If you are the kind of person who gets bored easily and need an overview before you will remember things; use this approach: Read the lessons, do copy and paste code where you can, and fly through the book to get an overview. Return to the beginning and do the lessons slowly and type all the code for practice and mastery. Repeat the lessons until you master them.**

Programming is learned best by doing it until you got it. Some people find that having an overview of the entire course and coming back to practice is a better approach. If you have a teacher or instructor for this course, then you should trust their experience and judgment. They will have specific objectives and a specific time frame planned for you. In this case you have the following choices:

1. **Listen to your instructor.**
2. **Fail the class.**
3. **Learn it your own way and gamble with your grades.**

The exercises provided should be repeated and altered at will until you reach a point of confidence and understanding. Don't just copy the code in the book, **play with it.**

Keep in mind that one of the primary purposes of this course is to give you some confidence in doing some simple programming while giving you an overview of Python.

You can dive into the deeper depths of every topic within this course as you feel ready to do so. To reach more advanced levels of programming see python.org's website; visit Python communities; advanced books; Python 3x or newer code exchanges; online resources, resources in Index C of this course, Python's documentation, python.org and www.toonzcat.com .

Other Uses for this course:

Of course, there are other uses for a book or course if you have a creative mind. My wife thinks books are where you put the hot frying pan so it doesn't burn the table. I once had an interesting experience when I taught my first class in China; that opened my eyes to other creative uses for a book:

Setting: ***One of the most famous Universities in China. Sometime after the Qing Dynasty. Daytime.***

I had just eaten some, not so safe local food, giving me a new definition to the term "fast food." I desperately searched for a bathroom on my way to class. Surprise one happened when I opened the door of the stall. I expected to see a friendly toilet; in its place was a porcelain lined hole in the floor. But this was a desperate situation; I had no time to contemplate neither the reasoning behind someone stealing the toilet nor the philosophy behind lining the hole in the floor with porcelain. With some relief everything quickly came out OK. Then came surprise number two. I looked around for toilet paper. It was at that memorable moment in my life when I discovered a difference in our two cultures. In China, you are expected to bring your own toilet paper. Caught with my pants down, I looked at the time and saw I was about to be late for my first class. I frantically looked around the cubical for alternatives.

I entered the classroom as the bell was ringing, just in time! I went to the podium, cleared my voice, and declared; "For this lesson we will go directly to chapter 2; chapter one has been chosen to serve a higher purpose."

Why the Strange Format?

Because strange, surprise, out of place, new, different, humorous, emotional, artistic, ugly, and downright weird experiences help us to learn faster. Oh, and of course the old tried and true method of seemingly useless repetition. Did I mention repetition? Our attention span is not much different than a child's in this respect. Sure, we have been beaten, threatened, and brainwashed into sitting like motion-less zombies in a classroom desk to pretend we are paying attention. In fact, those who move too much in some cultures are branded "hyperactive" and even drugged, (sorry "medicated"), to get them to sit still and stop interrupting those who are pretending to pay attention around them. When, in reality, we all have varying degrees of success in these vain attempts to continually focus, depending upon many factors such as; level of interest, emotions, the amount of sleep we had last night, and how many distractions there are in the room that may be more interesting. From the beauty in the corner to the fly buzzing around someone's head, our mind occasionally longs for something, anything, other than the teacher or the subject of the class to focus on. We even tend to exaggerate the significance of these distractions to prolong our escape from the zombie impersonations expected of us.

A short story for understanding the moral of this section;

Cartoons by Jody S. Ginther ©2013

A *little* bee flies into the window in a family home. The family members may or may not notice, but will generally react with mild amusement or fear. They will continue watching TV, terminate the bee, let it out a window, or calmly lock it in grandma's *room*.

The same bee flies into a public classroom. Now we have a major event! Girls screaming, people laughing, irrational ducks, leans, objects being thrown, and other acts of total anarchy. Chaos escalates until the bee is no longer in the room or is viciously murdered after being beaten half to death by a mad mob during it's futile attempt to escape.

The same event occurred. But, as you can see how in the second instance; the importance of the bee was magnified in the minds of our fellow zombies, as a desperate, welcome escape from their required payment of time and energy to attention.

Even the importance of a simple insect can radically increase as our minds become desperate to fly around real, imagined, and internal realities as a welcome escape from focusing on one topic too long.

SO, WHAT'S MY POINT?

Ok…I'll get to it;

Faster learning methods, *(in this case visual and unique, with tried and sometimes true methods),* provide my excuse for not formatting this course in a *boring, or I should I say; "traditionally correct"* format. Let's face it; unless you have one of the few whose brains that are wired differently, things like math and programming can be boring. Furthermore, it is more fun for the author, to present this stuff in a different way.

TIPS FOR FASTER LEARNING:

- YOU WILL LEARN FASTER IF SOMETHING IS INTERESTING TO YOU. Change is **important to maintain interest.**

- THE HIGHER THE EMOTIONS ARE WHEN YOU RECEIVE THE NEW INFORMATION, THE EASIER IT WILL BE TO REMEMBER. I CALL THIS "EMOTIONAL LEARNING." THINK OF FUNNY, INTERESTING OR EMOTIONAL THINGS WHILE LEARNING.

- VISUALIZATION IMPROVES RETENTION.

- PHYSICALLY *DOING* SOMETHING IMPROVES RETENTION. SO, TYPE ALL THE EXAMPLES MULTIPLE TIMES IF NECESSARY.

- REPETITION AND PRACTICE. REPEAT, REPEAT, AND THEN REPEAT.

- VERBALIZING WHAT YOU LEARNED CAUSES DEEPER UNDERSTANDING, (OR DEEPER DELUSIONS).

- PACING OF THE INFORMATION IS IMPORTANT. KNOW WHEN TO TAKE BREAKS AND WHAT INTERVALS TO REVIEW AND REINFORCE NEW INFORMATION. KNOW YOURSELF. SOME OF THE LESSONS ARE VERY SHORT AND SOME ARE VERY LONG. IF THEY ARE TOO LONG SPLIT THEM INTO SMALLER PARTS.

- SIMULTANEOUSLY STIMULATING NOT ONLY BOTH HEMISPHERES OF THE BRAIN, BUT DIFFERENT REGIONS OF THE BRAIN IMPROVES LEARNING. CLASSICAL PIANO MUSIC SUCH AS BACH'S GOLDBERG VARIATIONS, OR MOZART IN THE

BACKGROUND WHILE STUDYING CAN HELP. THE MUSIC MUST BE GENTLE, QUIET, AND NOT DISTRACTING. I FIND SOLO PIANO OR GUITAR WORKS BEST.

- **ALPHA BRAIN WAVE STATES; A RELAXED STATE OF MIND CAN HELP YOU LEARN FASTER. WHEN YOU FIRST WAKE UP OR ARE ABOUT TO SLEEP, YOUR BRAIN CAN RETAIN INFORMATION BETTER. GENERALLY, THIS IS BECAUSE YOUR BRAIN IS IN AN ALPHA STATE. WE HAVE ALL EXPERIENCED HEARING SOME SONG IN THE MORNING AND EVEN IF WE HATE THE SONG, IT IS STUCK IN OUR HEAD ALL DAY. WE EVEN GET DISGUSTED WHEN WE CATCH OURSELVES HUMMING IT. The first things you do in the morning and the last things you do before sleeping can more easily be retained.**

- **TEACH WHAT YOU KNOW. THIS FORCES DEEPER UNDERSTANDING. IF YOU CAN, FIND SOMEONE WHO KNOWS LESS THAN YOU TO EXPLAIN IT TO. WHEN YOU HAVE TO EXPLAIN SOMETHING, THIS REINFORCES THE INFORMATION. STUDIES HAVE SHOWN THAT TEACHING WHAT YOU KNOW IS ONE OF THE FASTEST WAYS TO RETAIN INFORMATION. THOSE YOU TEACH MAY ALSO POSE QUESTIONS YOU NEVER THOUGHT OF AND EXPAND YOUR OWN UNDERSTANDING BY FORCING YOU TO THINK ABOUT IT FROM DIFFERENT ANGLES.**

I won't go into the research and verification of these learning methods. You will just have to believe me or go do some research yourself. This course is about programming so we won't spend much time on the research end of faster learning. Just try these methods and you will see a difference in your ability to retain new information. Believe it or not.

I also find it is useful to read and record information into an mp3 file to use for passive learning. You can listen to the new vocabulary and concepts while driving, walking, exercising, etc.

Lesson 1: The Beginner's Tour

WHAT YOU WILL LEARN:

- **WHAT'S A PROGRAMMING LANGUAGE**
- **WHAT'S PYTHON AND WHY LEARN IT?**
- **IDLE AND THE SHELL**
- **IDLE COLORS**
- **WHAT ARE BLOCKS OF CODE?**
- **REVIEW**

WHAT'S A PROGRAMMING LANGUAGE?

All languages begin with an understanding of some new terms. We will begin with some essential words that you should know before we get started. If you know these, skip over them. If not, take a moment to learn them.

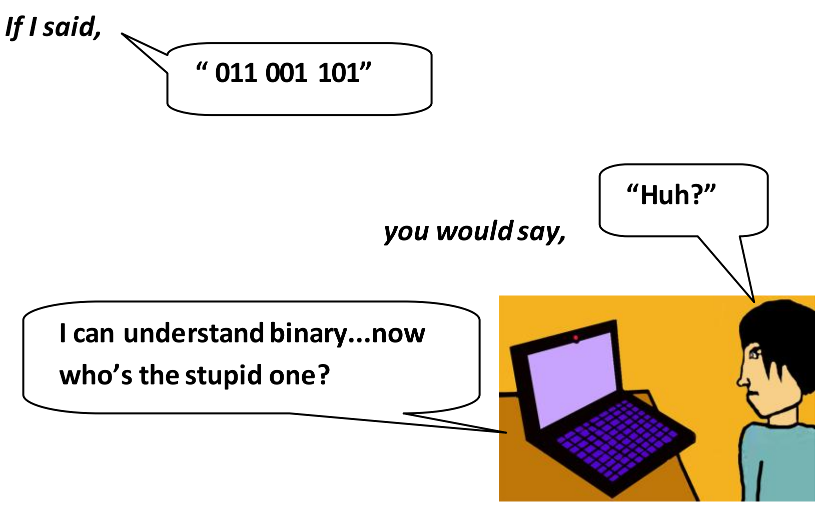
A programming language is a language you can use to communicate with a computer.

What are programs? ***Programs*** *are algorithms and source code packaged together to achieve some objective(s).* Programs do something for you. As programmers, we should think in terms of how to solve problems with good decision making logic to reach the results we want. What's an algorithm or a source code? ***Algorithms*** *are sets of instructions that tell the computer what to do.* Algorithms tell the computer how to reach a goal or objective. Ok, an algorithm is a set of instructions; then what is source code? The ***"source code"*** *is referring to all the instructions written in a particular language.* All the words, commands, secret symbols, and other stuff we typed or didn't' type into our program is the source code.

For example; in our programs we will write the instructions, (algorithms), in Python source code. You could think of a program as a book with a specific story and purpose; algorithms as the sentences and paragraphs; and source code as all the words, letters and symbols used in that particular book. In our "books," (programs), we will write our source code in the Python language rather than common English.

Note: Project files for this course are available at www.toonzcat.com

Programming *is the art and science of making the computer do what you want it to do by creating programs.* Computers are stupid and don't understand English. So, we have to use computer languages to translate what we say into Computerish. Actually, the language computers speak is referred to as a binary language. Binary language is a language based on two words; "on" and "off" represented by the numbers 0 and 1. Humans have trouble communicating in binary.



So, new computer languages where invented to make it easier for human brains to understand and be able to communicate with the computer.

Why so many computer languages?

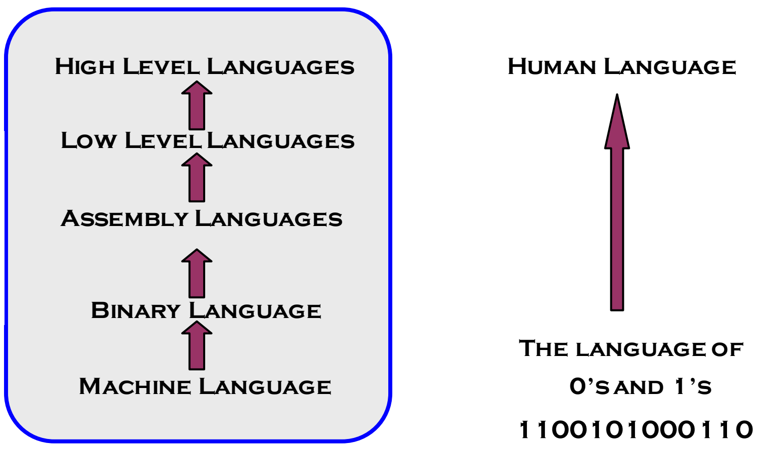
Different languages were designed for different purposes. Some are better at math, some are better at controlling computer hardware, and some are better for the internet. Python is a general purpose language. It can be used for many different purposes. Different computer languages are like different tools to a carpenter. You can still build a house with a hand saw and a hammer; but sometimes an electric saw and a staple gun would be better for a particular job. The world of programming is the same. You can survive using only one programming language, but you may get the job done faster and easier by having more than one tool to do the job.

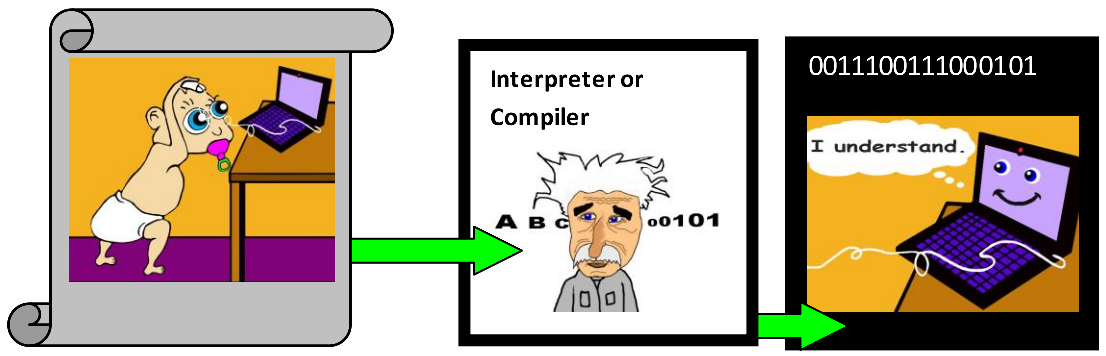
What's Python and Why Learn it?

Yes, it's a snake but Python is also the name of a computer language. Python is one of the easier computer languages to learn, yet it is very powerful. No programming language is truly "easy," for everyone in the beginning. Mastering anything takes time and effort. When Python has become easy to you; you have become a true Python master.

Python is known as a scripting language, (uses scripts), and is a high level language. A ***high level language*** *is a computer language that is closer to human language and easier for us to use than low level or machine languages.* High level languages also take care of many tasks like manipulating the memory of the computer for you. ***Low level languages*** *are used when the programmer wants more direct control over the machine he is using.*

This chart gives you a brief over-view of the levels of programming languages.





Interpreters and *compilers* both translate high level language code into low level binary code so the "machine" can understand it. However, they do the same job in different ways. *Why two different approaches?* Using a compiled language will generally result in faster running code. While using an interpreted language will generally result in less time for the developer to write and debug the code.

Here are some general comparisons:

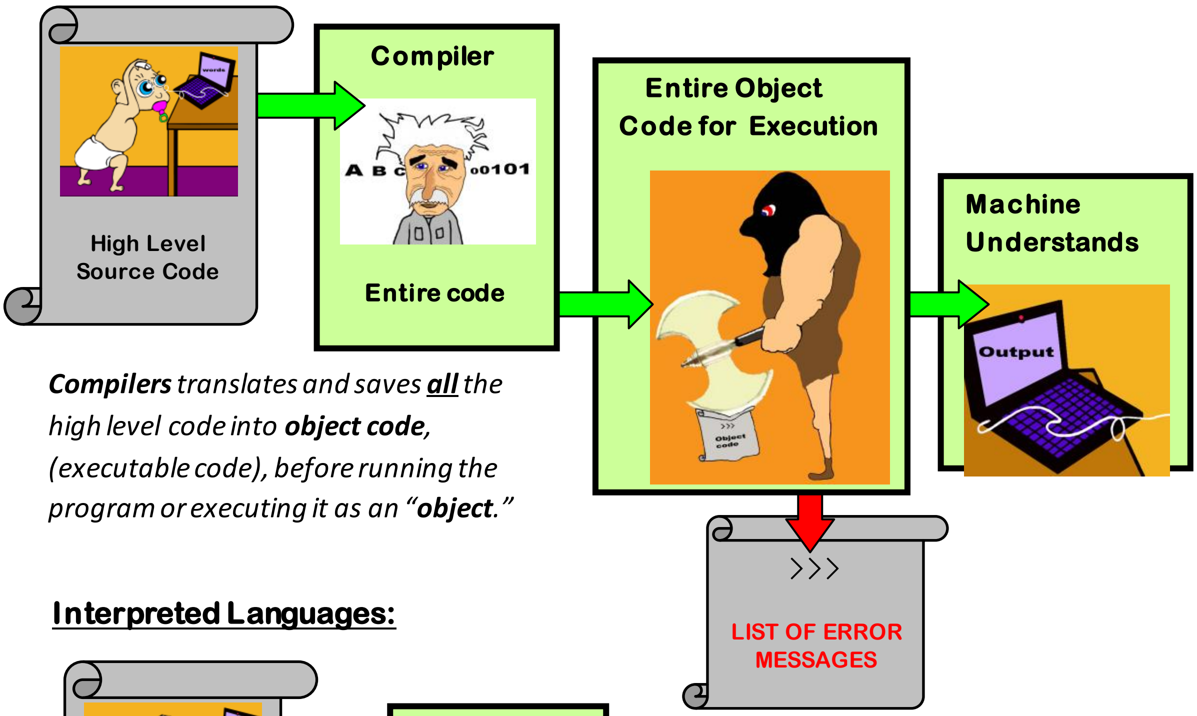
Compiler:

- translates entire code into an executable
- saves code as it compiles it
- provides a list of errors after everything is translated
- takes longer to process code and find errors
- final code runs, (executes), faster

Interpreter:

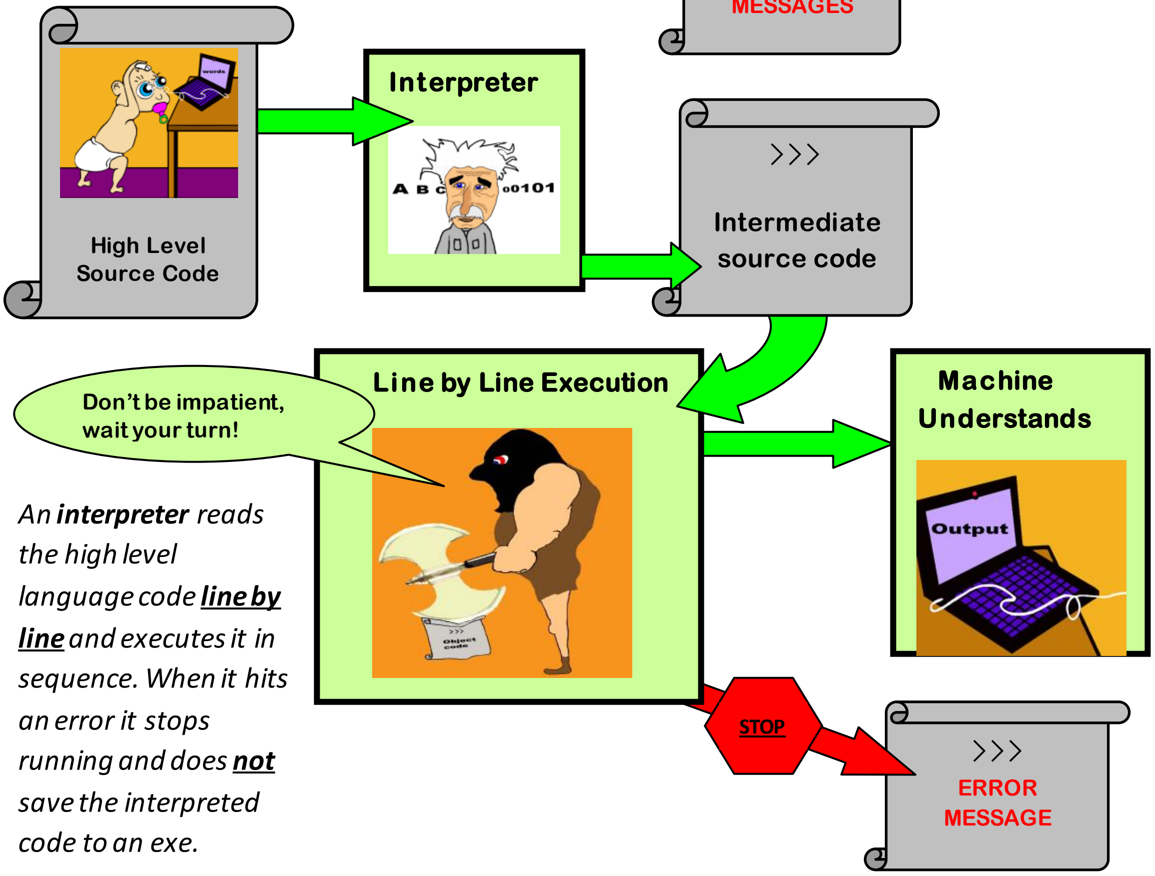
- takes statements one-by-one to translate and execute
- does not save interpreted code
- stops translation when an error is detected and displays the error
- faster in processing code and generally for the developer to find errors.
- final code may run, (execute) slower

Compiled Languages:



***Compilers** translates and saves **<u>all</u>** the high level code into **object code**, (executable code), before running the program or executing it as an "**object**."*

Interpreted Languages:



*An **interpreter** reads the high level language code **<u>line by line</u>** and executes it in sequence. When it hits an error it stops running and does **<u>not</u>** save the interpreted code to an exe.*

Python is an interpreted language. It does not need to be "compiled" before running it.

So what does this mean for you? It means you save a step and some time. You don't have to send your code through a compiler before you can run it and test it. Python even allows you to run small amounts of code in an "*interactive*" or instant manner as you write it.

Python simplifies some tasks in programming and is more readable to humans than many programming languages. Since code can be packages into moveable "*modules*" it can also save time by allowing you to reuse clones of code modules rather than rewriting them from scratch.

So, what is Python used for? Python is use for anything from games and data base programs to web applications. It is used from the New York Stock Exchange to NASA.

Python is considered a valuable tool and is used by many programmers throughout the world. Best of all, Python is easier to learn than most programming languages and is open source! So, let's get started!

Note: To read the history of Pythons development, refer to documentation at Python. org.

Getting and installing Python

Python is free. Download it from python.org. I used Python 3.2 for most of this course and played a little with 3.3. As you read this, a newer version may be available. I suggest choosing the latest stable version for your operating system. When using additional stuff like PyGame make sure it is compatible with the version of Python you are using. I primarily used Python 3.2 because it was thoroughly tested and compatible with the other software I used in these lessons. There were major changes made from Python 3 on, so backwards compatibility is limited. This course is not intended to be used with earlier versions of Python. Follow any special installation instructions you may need and check for the latest versions at:

http://www.python.org/download/

IDLE and the Shell

First, you can start Python in Windows by clicking on **Start>Programs>Python 3.x>IDLE**. Go ahead, try it...I'll wait here. ***IDLE** is software that helps you to communicate with the computer.* It is on the outside of the main program, just as a snail's shell is on the outside of the snail. Whack on a snail's shell and he will get the message;

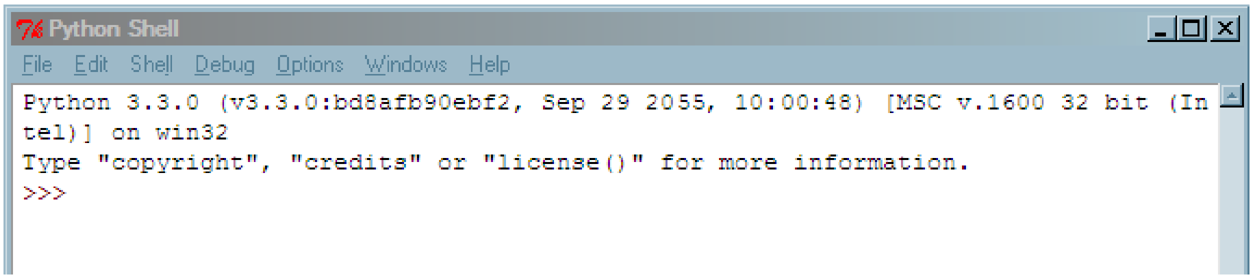
IDLE works in a similar way; enter commands into IDLE and it will translate and send the message to your machine.

IDLE acts as your *interpreter* and translates what you say into a language that the computer can understand. If you really want to know, IDLE stands for **I**neractive **D**eve**L**opment **E**nvironment. Why did they choose the L in the middle of the word, "development?" I have no idea and it's not important for the purposes of this book, so let's move on and get over it.

There are two windows to work from in IDLE. There is the **Edit Window** and the **Shell Window**. The Python Shell window will say, "Python Shell" at the top of the window, while the Edit window will say, "Untitled" and have a "run" command listed on the top menu bar. If Python starts in the Shell Window and you want to use the Edit Window, just choose **File<New Window** and it will open an Edit Window. If you want to use the Shell Window from the Edit Window go to the "Run" menu at the top of the window and choose "Python Shell."

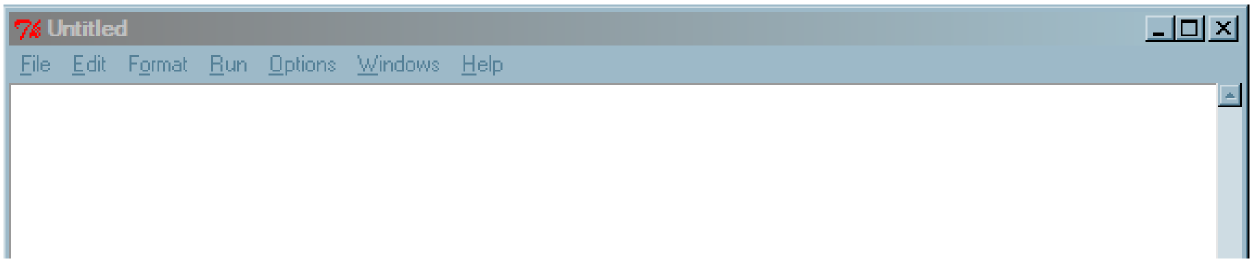
The ***Python Shell*** *is an interactive interpreter*. This means that when you press the enter key, it checks your source code and may give you some feedback. If you see (>>>), this is the first command prompt. It is letting you know the interpreter is patiently waiting for you to type something. If you see (…), this is the secondary prompt waiting for you to type something more.

The Python Shell looks like this:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2055, 10:00:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

The Python Editor looks like this:



There are other editors that you can use for programming but to keep things simple in this book we will use IDLE for nearly everything. Besides, it's already packaged with Python. Remember that if you chose to use another text editor; you should stick to specific or simple text editors. Editors like notepad rather than one that will mess up your code by adding textual elements or applying unwanted formatting, (like Microsoft's Word program). We will discuss other useful editors later in the course.

Why are there two windows in IDLE and how do I use them? Python allows you to work in *script mode* or in *interactive mode*. What's the difference?

Script Mode *is great for writing programs you can save and run later. It is generally used for the final product*. You are running Script mode when you are in the Edit window.

Interactive mode *is for testing and trying small ideas quickly*. You are running in interactive mode when you are in the Shell window. Most people use both of these together. Script mode for working on their main program and interactive mode for trying new ideas; in the same way you would use a piece of scratch paper.

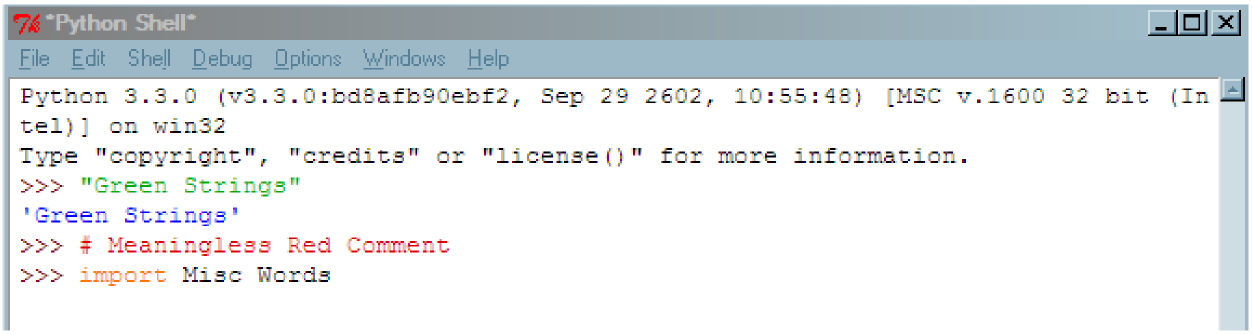
A third function of IDLE is that it is also a debugger. You can find the debugger button at the top of the Shell window. What's a debugger? It's a program to help you find bugs. No not the one crawling up your leg or living in your hair; but the bugs, or problems, that are in the code we are working on. Why are computer problems called bugs? Computer history tells us that one of the first major computer crashes was caused by an actual bug getting toasted on the circuit board of a computer; hence they named problems in programming computers as bugs. We will go deeper into that story later.

IDLE Colors

Did you notice that IDLE changes the colors of your text? What's the meaning? Let's look at a list of some of the most common syntax colors and their meanings in IDLE.

Common Python syntax colors:

Keywords	orange
Strings	green
Comments	red
Definitions	blue
Misc. Words	black

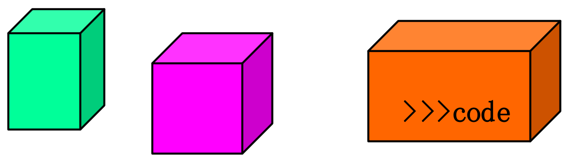


```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2602, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> "Green Strings"
'Green Strings'
>>> # Meaningless Red Comment
>>> import Misc Words
```

You needn't memorize these at this point, but knowing the meaning of the colors can help you see clearly where you typed something wrong.

If you were trying to type a string and it's not green, you probably forgot the quotation marks or did something else the computer didn't like. Don't worry, I will tell you what a string is later. The colors can also be changed according to your personal preferences and may vary in different editors.

WHAT ARE BLOCKS OF CODE?



While we are looking at the windows, how is your text arranged? The text is arranged in *lines, groups, and blocks*. A ***block** is just a group of code that goes together*. Like a city block, it can be divided into smaller groups like houses on the block, cars on the block, people, dogs, lions, cockroaches; etc. on that block. This concept is important to tell the computer how to read and follow your code. To a computer, arranging your code with the proper grouping of lines and blocks is like a map that says; "first do this, second do that, or repeat this part. " It can also give directions like a map within your program by saying;

"After doing this block go to block x and do... that."

Blocks are one way we dictate the running order in programming. You can have blocks in blocks just as you can have groups in groups. You may have a group of students under 100 years old. But, within that group you may have another group called "girls." Inside of that group you may have another group or block called "the names of girls with green hair."

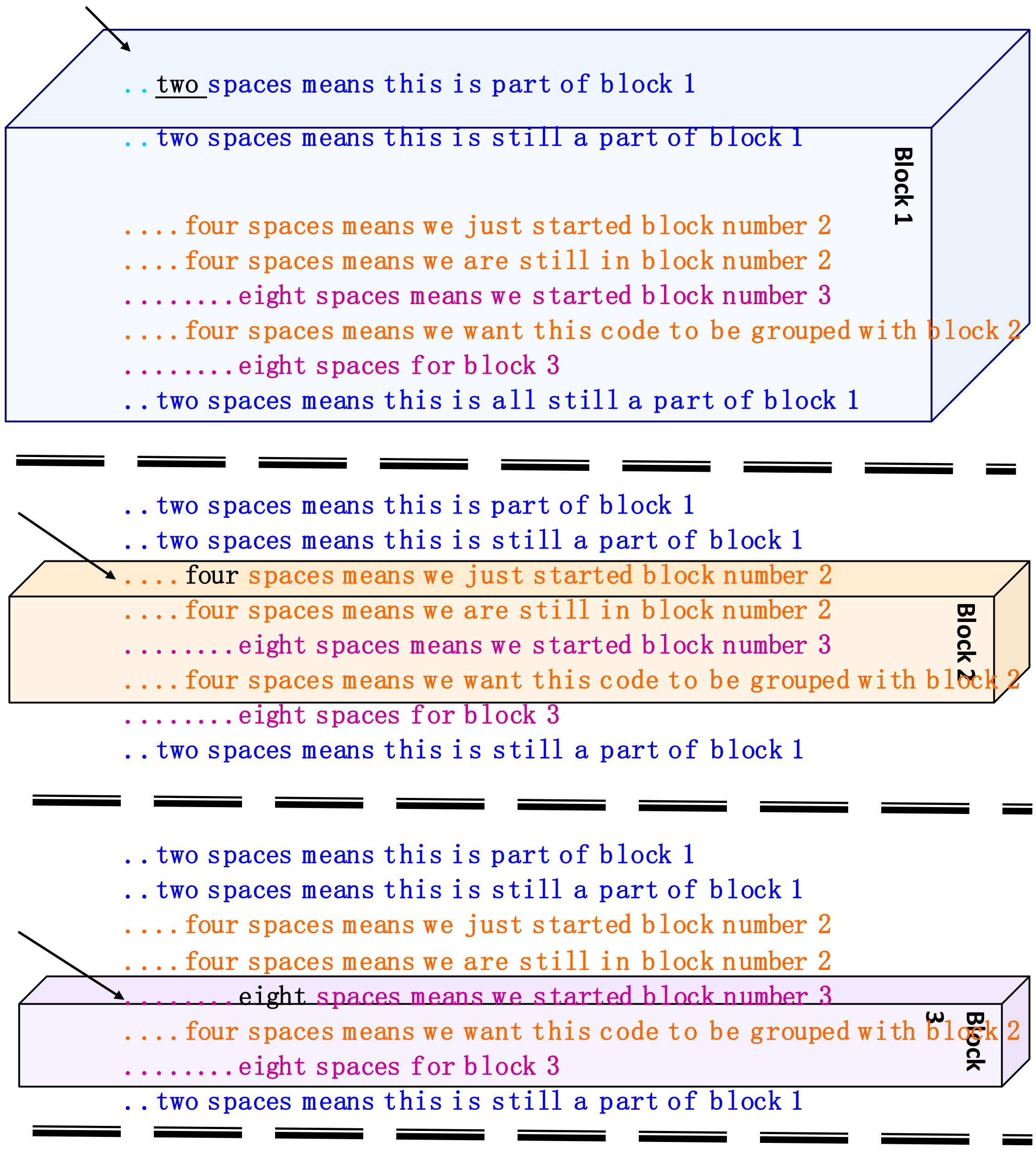
Blocks make it easy to keep things or instructions in our code together in their correct group. This helps us refer to them and to direct the computer to use that group of instructions, in the order we want the computer to read them. Blocks are defined by the number of spaces used to indent each line of code. In the following examples I will use dots to show you clearly how many spaces would be in the code. In this lesson; "**...**" refers to three spaces. Remember to use spaces, not dots, when you type your code into Python.

***Note:** The color code in the following examples is <u>not</u> a part of IDLE's color scheme. These colors are used to emphasize what parts of a block belong together.

2 spaces ..students under a hundred years old ***(Block 1)***
4 spaces ....girls ***(Block 2)***
8 spaces ....girls with green hair ***(Block 3)***

<u>**The number of times we indent**</u> helps the computer know ***how we are grouping the information.*** (Hang in there, a few more ideas and you will make your first game).

Here are some visual examples:



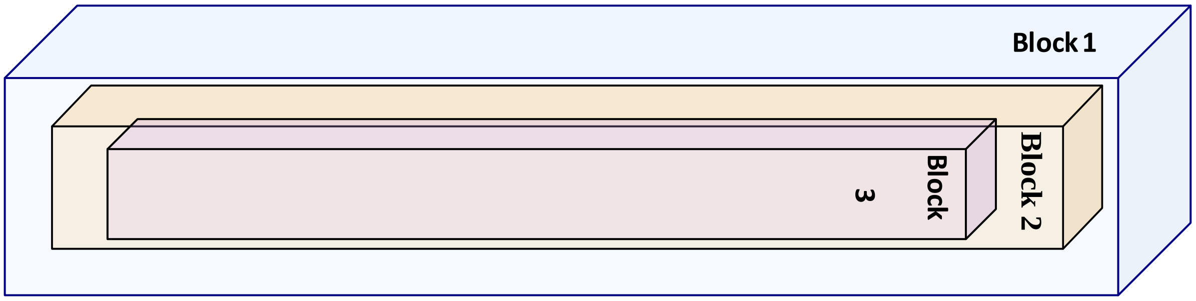
```
..two spaces means this is part of block 1
..two spaces means this is still a part of block 1

....four spaces means we just started block number 2
....four spaces means we are still in block number 2
........eight spaces means we started block number 3
....four spaces means we want this code to be grouped with block 2
........eight spaces for block 3
..two spaces means this is all still a part of block 1
```

```
..two spaces means this is part of block 1
..two spaces means this is still a part of block 1
....four spaces means we just started block number 2
....four spaces means we are still in block number 2
........eight spaces means we started block number 3
....four spaces means we want this code to be grouped with block 2
........eight spaces for block 3
..two spaces means this is still a part of block 1
```

```
..two spaces means this is part of block 1
..two spaces means this is still a part of block 1
....four spaces means we just started block number 2
....four spaces means we are still in block number 2
........eight spaces means we started block number 3
....four spaces means we want this code to be grouped with block 2
........eight spaces for block 3
..two spaces means this is still a part of block 1
```

Whatever the number of spaces you choose, keep it consistent so you don't get confused.
Why do we care about blocks and grouping programming statements? Ah, I'm glad you asked, but I won't tell you until later.

Each block of code can reside and work within another block of code.



Review: *(Yes, you should know these terms.)*

Before we get to the fun parts we have to speak the same language. Even things like going to the bathroom require special vocabulary like; "Toilette Paper!" "Open the window!" "Did I really eat that?"

Quiz 1:

Answer the following questions on a separate piece of paper or in a document if you prefer. Check your answers on the next page.

1.1 Define the following terms:

1. Algorithms
2. Block
3. High Level Language
4. IDLE
5. Interactive Mode
6. Programming
7. Python Shell
8. Programs
9. Script Mode
10. Source Code

1.2 Questions:

1. What's the difference between an interpreter and a compiler?
2. Which one does Python use?
3. Why is indentation important?

Vocabulary

Algorithms *are sets of instructions that tell the computer what to do.*

Block *is just a group of code that goes together.*

High level language *is a computer language that is closer to human language and easier for us to use than low level or machine languages.*

IDLE *is software, (an editor), that helps you to communicate with the computer.*

Interactive mode *is for testing and trying small ideas quickly.*

Low level languages *are used when the programmer wants more direct control over the machine he is using.*

Programming *is the art and science of making the computer do what you want it to do by creating programs.*

Programs *are algorithms and source code packaged together to achieve your objective(s).*

Python Shell *is an interactive interpreter.*

Script Mode *is great for writing programs you can save and run later. It is generally used for the final product.*

Source Code *includes all the algorithms or instructions written in a particular programming language*

Quiz 1 Answers:

1.1 Refer to the vocabulary terms chart above.

1.2

1. ***Compilers*** *translate* ***all*** *the high level code into* ***object code*** *before running the program. An* ***interpreter*** *reads the high level language code* ***line by line*** *and executes it in sequence.*

2. Python is an interpreted language, therefore it uses an interpreter and has no need for a compiler.

3. Indentation tells the computer how to group the code into blocks that will be run in a particular order.

Lesson 2: Let's Begin Programming!

What You Will Learn:

- Your First Program
- Variables
- Operators
- Key Words (To use or not to use)
- Boolean Data Types
- Introducing Strings
- String Placeholders and Escapes

You're First Program!

You are now a programmer, or will be shortly. Don't be intimidated by all the new geek words within each lesson. You will be speaking Geek as well as typing in Python faster than you think, (unless, of course, you are delusional.)

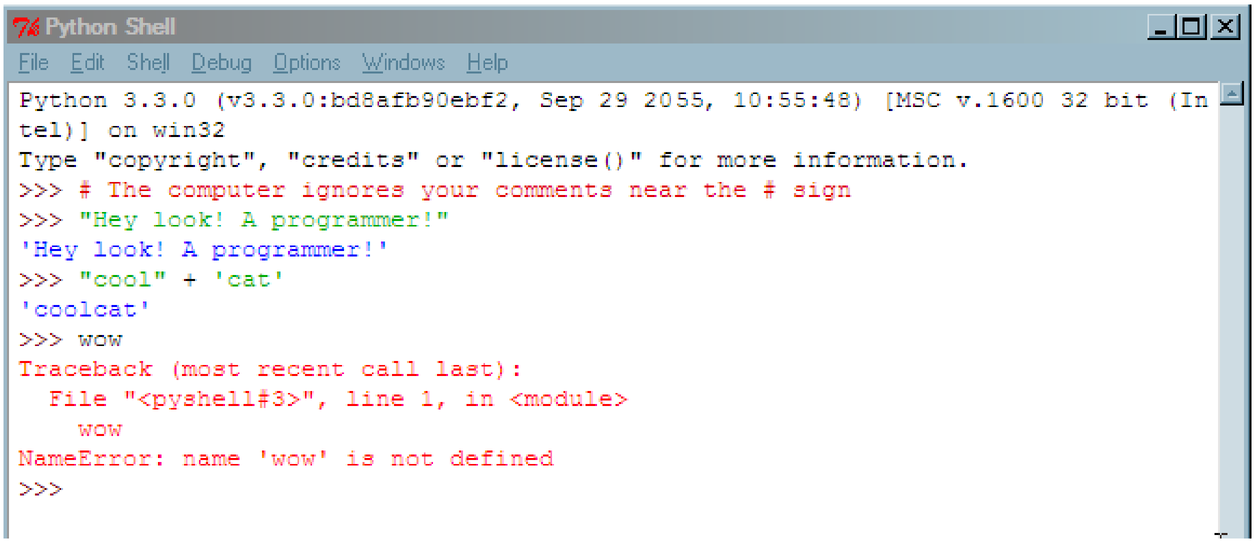
Ok, let's start playing…err… programming. Open your Python Shell window to type some things. Type the following code exactly as you see it. ***Then hit the enter key after each line.***

```
# the computer ignores comments near the # sign
"hey look a programmer"
"cool" + 'cat'
wow
```

From this point on in your life, you have to be a fanatic about details and a perfectionist about typing. Every space, dot, symbol, upper or lower case, can have an impact on your source code. If something doesn't work, go back and look at your typing carefully.

Most errors by beginning programmers are in the syntax of their code. This is why it is important that you don't just copy and paste code, but you actually type it in for practice and perfection of your programming skills. Train your eye and your fingers, (or both eyes if you have two).

If you entered these correctly, it should look like this:

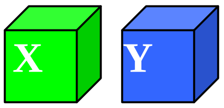


```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2055, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> # The computer ignores your comments near the # sign
>>> "Hey look! A programmer!"
'Hey look! A programmer!'
>>> "cool" + 'cat'
'coolcat'
>>> wow
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    wow
NameError: name 'wow' is not defined
>>>
```

If you type the # key in front of some text, the computer ignores the text. This is useful for adding comments in your programs that will not be misunderstood as instructions by the computer; or if you just feel like being ignored. Ah, our first error message. Get used to these because they are the programmer's constant companion. In time, you will see them less and understand they do have a purpose other than raising your blood pressure.

We will be explaining more about this code shortly; for now I just wanted you to type something and get the feel of the shell.

VARIABLES



Have you ever asked; "When am I going to use this kind of math?" Now that you are a programmer you should remember some simple math concepts to make your life easier. Don't worry, I'll be brief. You don't have to remember complex math at this point, but higher levels of math will definitely increase your programming skills. For now all you need to know is the world of math has animals called "*operators*" and "*variables*." In programming we use these same mathematical concepts and math-like logic when designing algorithms.

Variables** are like little like boxes or containers to put different things in*. In math your teacher may have told you that 1 + *x* = some other number. The 1 is an ***integer, *(a complete number as opposed to part of a number like ½)*, and the *x is a variable*. In programming, you get to name your variable anything you want. You can create an imaginary box with anything you want to put in it, and define/label that imaginary box, (or variable), by any name you choose.

This means a variable in programming can refer to or contain integers, strings, and some other data creatures you will be meeting as we continue on this path into the world of Python. Let's try it. Let's imagine a box of chocolate. We want to tell the computer that the box labeled "chocolate" has happiness and joy inside of it. To do this we use the = sign to define what our variable means for the computer to understand us.

In the Python shell window type:

Chocolate = **"happiness and joy"**

Hit the enter key.

Now, let's type the word without quotes;

Chocolate

Hit enter.

Your computer should reply, **'happiness and joy'**

You just *defined* a variable and then *called* it. This is very useful in programming. You will constantly be teaching the computer how to think as you write programs and will define things for it.

Variables** are chunks of data stored in the computers memory*. There are generally three types of data stored in variables. Variables can be in the form of ***integers or in a ***string*** as mentioned previously. Another type of data, called a ***float***, refers to the *non-whole numbers like decimals*. Remember to use the = sign to assign a variable. If you want the computer to think the variable *x* means 5 is in the variable box named "*x,"* then you type:

```
x=5
```

Now the computer holds a 5 in memory and when you type or call an x, it tells you '5' when you hit the enter key. *What if you want to name a variable now, but you want to assign a value or put something in it later?* One way is to assign the no value, or "None." You could do it this way:

```
x=None
```

You can then "overwrite" or replace the value "None" later. There are times when you write code but don't know what value will be placed in a variable. All you have to do to delete and replace the word "None" with the value, (or to overwrite it); is type it with the new value you want to assign to it, (in this case "23):

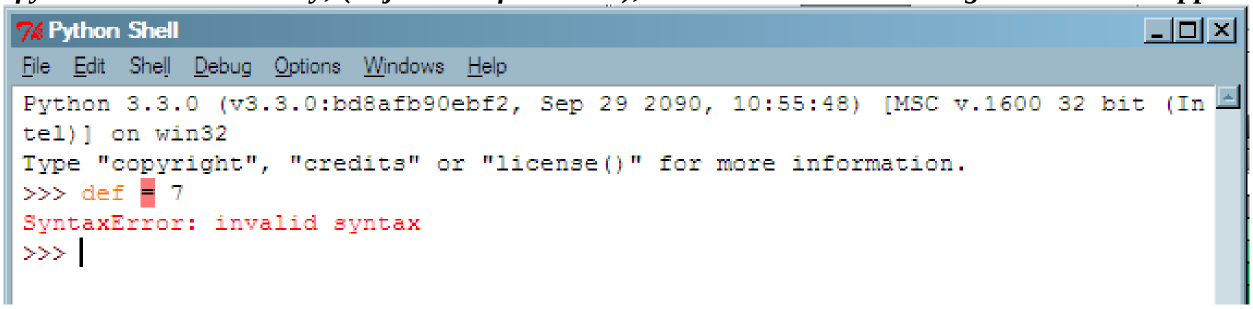
```
x=23   #(overwrites the old x value)
```

KEYWORDS: TO USE OR NOT TO USE?

You can name a variable almost anything and use symbols like the _ underscore. But there are some rules. You can't use special key words that Python understands as having special meanings. Don't use these words for variable names:

```
and   continueexcept     global   lambda     pass     while
as    def       False       if       None    raisewith
assert    del      finally      import    nonlocal  return yield
```

If you should accidentally, (or just out of rebellion), use these words something like this will happen:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2090, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> def = 7
SyntaxError: invalid syntax
>>> 
```

"`SyntaxError: invalid syntax...`" is Python's ways of saying, "Hey, you don't know what you are talking about and neither do I! Speak Python!" Don't take it personal, as a programmer you will get used to these insults. You will find your computer has an attitude. Don't begin a variable name with a number. The name "34th_apple" would not be accepted by Python. Underscores are allowed. There are also other illegal characters you can't use because they are used for other things in Python. Like the dollar sign, ($).

Python is case sensitive, so remember when you used capital letters. It also pays attention to blank spaces.

Other than these minor rules, you can name a variable anything you want. To sound more like cool geeks, we don't always call them "names," we sometimes call them "identifiers." Remember that ***identifiers** are just the names we give to our containers.*

```
people_list = [man, woman, child]
```

Operators

***Operators** do something, like add, multiply, divide, subtract, or compare.*
Notice that we already used the = sign to assign an identifier. So, we can't use the equal sign as an operator. Instead we must use == to mean the; "equals" symbol in normal mathematics. *What other useful things can be done with operators?* In the Python shell type:

```
"words words words words words"
```

Hit enter and type:

```
"words  "  *10
```

Hit enter.

Which is easier? If I knew this in primary school I could have really saved countless hours and gone home at the same time as the other kids. I could have just written:

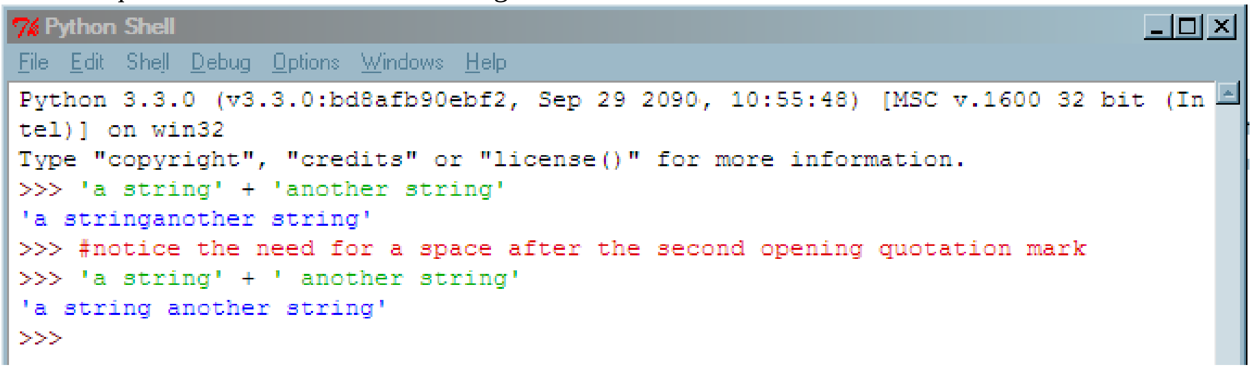
"I will not run on the desks nor refer to my teacher as something I saw floating in the toilet." * 1000

Of course, at that time my teachers preferred to see my handwriting rather than a print-out. Operators are life saving tools for the programmer. A good programmer in Python will rarely type the same code more than once. As you guessed by now, the asterisk * symbol is used to express multiplication and to repeat things. You could use it to repeat a string or tuple for a specific number of times. Remember to add a space at the end of "words " before the closing quotation marks or your 10 words will look like one long one.

+ for addition
- for subtraction
/ for division
*** for multiplication**

These are the most common operators.

You can also use operators with strings of data. Try it. Make two strings with any words you want between quotation marks and add them together. ***Like this:***



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2090, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 'a string' + 'another string'
'a stringanother string'
>>> #notice the need for a space after the second opening quotation mark
>>> 'a string' + ' another string'
'a string another string'
>>>
```

We also have ***comparison operators*** for…you guessed it; *for comparing things!*

Here are some of those:

> for greater than
< for less than
<= less than or equal to
>= greater than or equal to
= = equal to
! = not equal to

Remember that we already used the = symbol to define or tell the computer the meaning of our variables. So we don't confuse our little computer, we must use double equal signs "= =" to express the traditional meaning of "equals to."

Now try some of these until you get bored. Any math expression will do. But wait! Use only integers; no decimals at this point.

Due to the way a computer deals with ***floating point numbers***, *(numbers with a decimal point)*; you won't always get the same results as a calculator. Use a **"."** in all floating point numbers to solve this problem, (1.0, 0.234).

For now, remember that Python is not your math teacher; although they may look similar. In math, 7.0 is still an integer because numbers like 0, 5, 177, are integers. But computers are not so clever, so in Python the number 7 is an integer but since 7.0 includes a decimal point; it's called a *floating point* number.

Logical Operators

The *words "**and, or,**" and **"not"*** are ***logical operators***. Logical operators are for comparing and establishing relationships. *A **simple condition** is a comparison that only uses two values:*

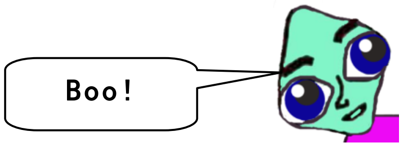
9<19

*A **compound condition** is a comparison using more than two values:*

x < 10 and x>5

These logical operators generally mean the same thing in programming as they do in English.

BOOLEAN DATA TYPES



What's a ***Boolean****, some kind of scary alien?* Wrong, you and the computer often need to know if something is true or not. *A* ***Boolean*** *data type is referring to two possible values;* ***True or False***. A computer is an electric machine and really only understands two thing; on and off. Binary is the computer language based on the use of 1 for on and 0 for off. Although reading 1001100011110000011110000 is easy for a computer, it is not so good for humans. As we mentioned earlier, this is why we use programming languages like Python to do the work of communicating with the computer on the machine level. But, we should still keep in mind that everything in our logic and programming will come down to two values we call "True" and "False." True is 1 and false is 0. To think like a computer, always relate your code to these values. A 0 in Python is false, while 1 or any other integer is true.

Boolean expressions are sometimes called *conditional expressions* because they *are based on the condition that something is either true or false.*

But, what exactly is an *expression* in Python? Shorty you will also run into the word, "*statement*." You know what they are in English but in programming they have special meanings. *An* ***expression*** *is any combination of values, (numbers), that can evaluated into another single value, (number).* This is just a fancy way of saying; " In 3 + 2 =5 is an expression." *A* ***statement*** *can be instructions that evaluate into a single value or a expressed fact,* (just as in English). If you wrote a line in your game like: "If the spaceship hits an asteroid, they both explode." This would be a statement.

Now that you are a programmer, your whole life will be about the pursuit of truth, (or not). You will be converting every problem and task into true, false, and logical sequences.

Let's play with some of this new knowledge. Open IDLE and type the following, but feel free to play with your own ideas after you try these:

8>3
Hit enter.
Ah, the computer agrees, it says, "**True**."

3>8
Hit enter.
Hey! The computer just said, "**False**."

I told you the computer has an attitude! Finding Boolean values of "True or False" is vital to the programmer.

To see if and expression is true or false you can compare the values in the expression using "==":

```
7==7
>>> True
```

```
7==4
>>> False
```

Introducing Strings

If you noticed I typed the words **"hey look a programmer"** in quotation marks. Later I used; **' '** single quotation marks and got the same exciting result. When we hit enter, the computer just repeated what we typed back to us. It printed the characters as output on the screen. *A sequence of characters, words, or sentences like this one is called a **string**. When we use single or double quote symbols to tell the computer what is in our string, we call these quotation symbols "**delimiters**." We tell the computer that we are entering or ending a string by using single or double quotation marks.* The computer don't care which kind you use as long as you are consistent in pairing them. We can now say we have *declared* or "***delimited***" the string, (a fancy way of saying we put it in quotation marks). You did what to that string? Yes, you delimited it and there is nothing to be ashamed of. In IDLE **strings are green** and **the output is blue**. The **error message is red.**

You can also add strings together using a math operator. We will talk more about that in a moment. Putting the string "cool" with 'cat' produced **'coolcat'**. If I want a space between them when they are added together, I should add one in my string; **"cool "+"cat" or "cool"+ " cat"** would generate **'cool cat'**, because in the first one I added a space after "cool" and in the second one I added a space before "cat."

What happened when I typed, "wow?" The computer said;

"Blah blah blah…is not defined. "

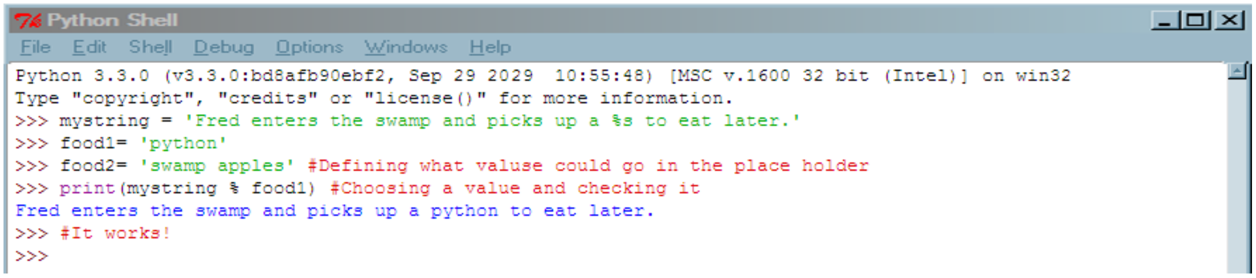
This is the computer's way of saying; "huh? I don't understand." Python attempts to give you an idea of what went wrong. When I typed the word, I did not include it inside quotation marks to tell the computer that I was entering a string. So, the computer went, "Huh?" "Not defined" means that you did not tell the computer the meaning of "wow". You did not *delimit it* nor did you *define* it for the computer by putting it into quotation marks.

String Placeholders and Escapes

There are times in programming that you need to have an empty space in a string to place something in later. It is like leaving an empty blank in a sentence to fill in later.

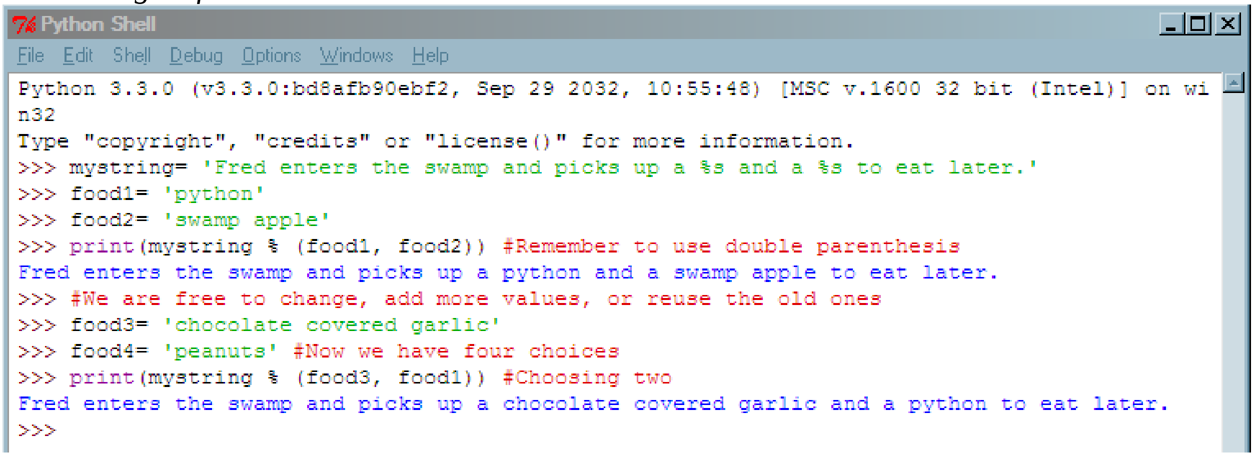
Let's say you are making a game and your character has to pick up and object.
You are not sure of which object at this point so you need to leave that part blank in your code. You will need to express something like; "Fred enters the swamp and picks up a _____ to eat later." We left a blank where we will place a word or value later. *A symbol in our code that indicates a blank value is called a **placeholder**. The position of the placeholder in a string can be indicated by using %s instead of a blank line. You use % to tell the computer what value(s) you want to put in that placeholder.*

Let's have a look at the code:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2029  10:55:48) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> mystring = 'Fred enters the swamp and picks up a %s to eat later.'
>>> food1= 'python'
>>> food2= 'swamp apples' #Defining what valuse could go in the place holder
>>> print(mystring % food1) #Choosing a value and checking it
Fred enters the swamp and picks up a python to eat later.
>>> #It works!
>>>
```

We can reuse our string and add more values. We can also add multiple placeholders in the same string as follows:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2032, 10:55:48) [MSC v.1600 32 bit (Intel)] on wi
n32
Type "copyright", "credits" or "license()" for more information.
>>> mystring= 'Fred enters the swamp and picks up a %s and a %s to eat later.'
>>> food1= 'python'
>>> food2= 'swamp apple'
>>> print(mystring % (food1, food2)) #Remember to use double parenthesis
Fred enters the swamp and picks up a python and a swamp apple to eat later.
>>> #We are free to change, add more values, or reuse the old ones
>>> food3= 'chocolate covered garlic'
>>> food4= 'peanuts' #Now we have four choices
>>> print(mystring % (food3, food1)) #Choosing two
Fred enters the swamp and picks up a chocolate covered garlic and a python to eat later.
>>>
```

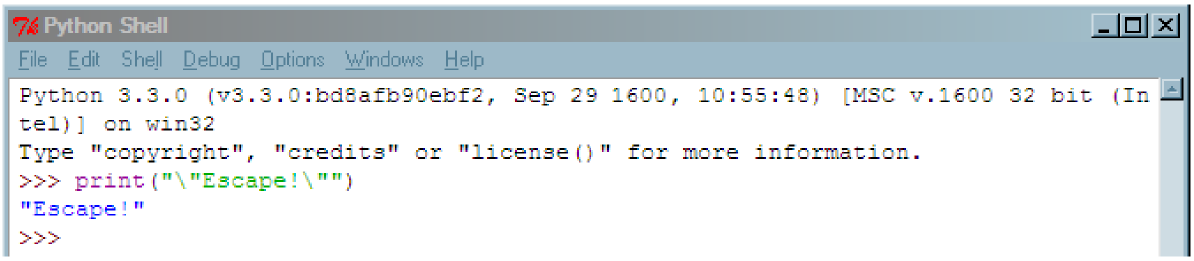
Sometimes you may want to use punctuation marks that Python sees as having another meaning. As you have seen, programmers assign other meanings to traditional characters.

True and totally useless story: I once asked two students to tell their meaning of the word; "love." One student rolled up her eyes, cooed and softly whispered, "Wonderful."
The next student stood up, violently swung a fist in the air and yelled; "Pain! Death! Misery!"
The same word; but two people had two very different meanings.

Just as two different people can see different meanings in the same word; Python must also be told which meaning we want to use if we have more than one.

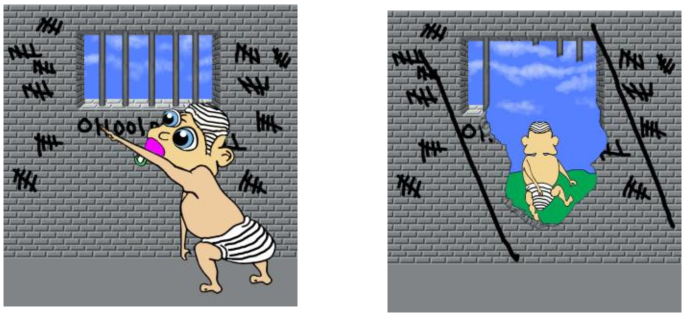
Now to escape! There are escape characters in Python to tell the computer to ignore the Python code meaning of the character and see it as normal punctuation.

The ***escape*** *character tells Python to stop or continue to read the code through the eyes of Python.* In this case we will use a pair of backslash, (\"x"\), escape characters to stop and then to continue Python thinking.



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 1600, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("\"Escape!\"")
"Escape!"
>>>
```

Without the escape characters we would only see the word without the quotation marks in our output.



Remembering the escape characters can be very useful.

A single backslash " \ " can be placed at the end of a line of code as a *continuation character*. This tells the computer to treat the continued line as a single line:

```
>>>stuff = "This is a very very very long string that I typed here \
to make my point. "
>>>print (stuff)
This is a very very very long string that I typed here to make my point.
```

```
>>>stuff = "This is a very very very long string that I typed here \n\
to make my point, but in this case I want two lines. "
>>>print (stuff)
This is a very very very long string that I typed here
to make my point, but in this case I wanted two lines.
```

To break a line of text and start another line use **"\n\":**

If you ever find yourself in a situation where you need to use a lot of backslashes, (probably won't happen until you get deeper into programming), you may have to tell Python to store all your backslashes as they appear. Otherwise the backslash will have a different meaning to Python and it will eliminate some of them as in the examples above.

You may need to do this if you want to output a DOS path in Windows. Messing with the backslashes in C:\program files\put it here\folder\ would not be a good thing. For this we use an "r'' to indicate our string is a *raw string literal*. Yeh, I know, just a weird way to say, "read and store it literally/as it is written." What's a "***string literal***?" A string literal is just fancy way to say; "*text or values between quotation marks for Python to ignore*." Notice in the following example how \n\ is ignored and printed as you see it in the text.

```
>>>stuff = "This is a very very very long string that I typed here \n
to make my point, but in this case I want two lines."
>>>print (stuff)
This is a very very very long string that I typed here
to make my point, but in this case I want two lines.
```

Review:

Quiz 2:

Answer the following questions on a separate piece of paper or in a document if you prefer. Check your answers on the next page.

2.1 Define the following terms and check your answers on the next page:

1. Boolean
2. Comparison operators
3. Compound condition
4. Delimiters
5. Float
6. Floating point numbers
7. Identifiers
8. Integer
9. Logical operators
10. Operators
11. Simple condition
12. String (not the one your cat plays with)
13. Variables

2.2 Practice Exercises

1. Print to output the following sentence, including the punctuation marks:
2. " This is a @#%$ sentence!, (Pardon the foul language)."

3. Create a string with a place holder and output the string with the value "7" in that placeholder.

4. What will you see if you type in "3>1" and hit enter?

5. What do I use to tell Python that I want multiple lines of code treated as a single
    a. line?

Vocabulary

Boolean *data type is referring to two possible values;* **True or False**.
Comparison operators *are for comparing things, (<, >, ==, !=, etc.).*
Compound condition *is a comparison using more than two values, (x < 10 and x>5)*
Conditional expressions *(also called Boolean expressions), are based on the condition that something is either true or false.*
Delimiters *defining quotation marks to tell the computer we are entering a string.*
Escape *character tells Python to stop or continue to read the code through the eyes of Python*
Float *non-whole numbers like decimals..*
Floating point numbers *numbers with a decimal point*
Identifiers *are names*
Integer *a complete number as opposed to part of a number like ½*
Logical operators *the words "**and, or,**" and "**not**"*
Operators *do something, like add, multiply, divide, subtract, or compare.*
Placeholder *(%s) a symbol in our code that indicates a blank value*
Simple condition *is a comparison that only uses two values, (9<10)*
String *a sequence of characters, words, or sentences in quotation marks.*
Variables *are like little like boxes or containers to put different things in.*

2.2 Practice Exercises (Answers)

1.
```
>>> print("\"This is a @#%$ sentence!, (Pardon the foul language).\"")
"This is a @#%$ sentence!, (Pardon the foul language)."
```

2. *Anything similar to:*
```
>>> mystring='I think I ate %s pizzas today.'
>>> pizzas = '7'
>>> print(mystring % pizzas)
I think I ate 7 pizzas today.
```

3.
```
>>> True
```

4. Use a continuation character, (a single backslash, \), in your code.

Lesson 3: Your First Game!

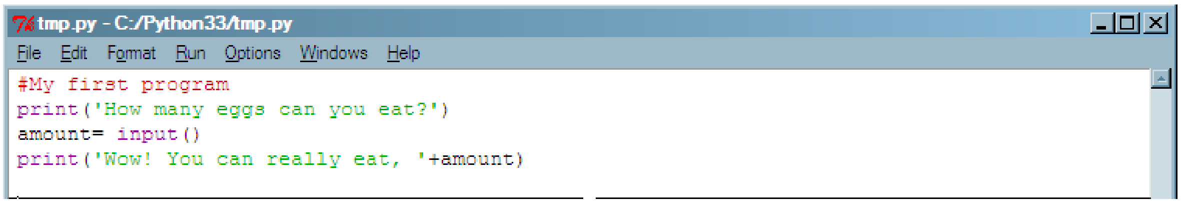
WHAT YOU WILL LEARN:

- **SAVING YOUR PROGRAM**
- **THE MEANING OF THE CODE**
- **CONDITIONAL STATEMENTS**
- **LOOP FOR A "WHILE" STATEMENTS**
- **GETTING LOOPY (MORE ON LOOPS)**

SAVING YOUR FIRST PROGRAM

To help you understand your first game you will write a smaller program first. Now you will work in the editor window and learn how to save your file. Open the *editor window*, ***not*** the *shell window*, and type:

```
#My first program
print( ‘How many eggs can you eat?’ )
amount=input()
print( ‘Wow you can really eat,  ‘ +amount)
```

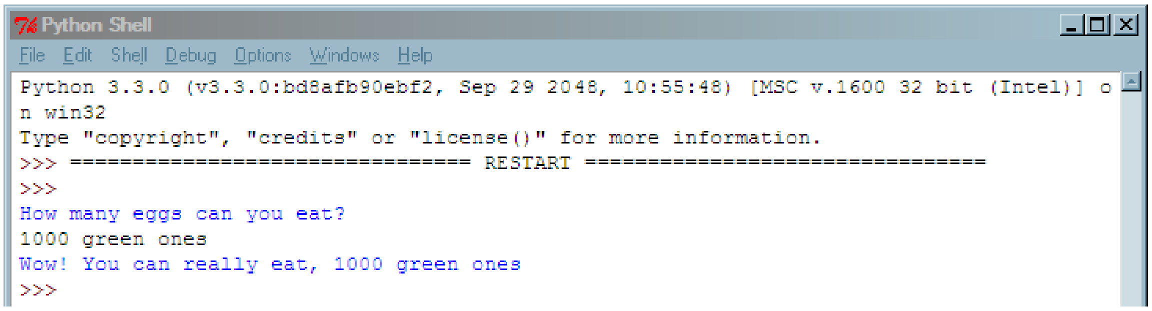


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
#My first program
print('How many eggs can you eat?')
amount= input()
print('Wow! You can really eat, '+amount)
```

Save your program and give it a name, but add .py after the name you choose. Then close Python. Reopen IDLE and go to *file*, then *recent files*, and choose your program name.
Go to the run button at the top of the edit window and choose "Run module."
When it asks you, "How many eggs can you eat?" type a number or a word. *I typed "1000 green ones"*.

It should respond and look something like this:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2048, 10:55:48) [MSC v.1600 32 bit (Intel)] o
n win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
How many eggs can you eat?
1000 green ones
Wow! You can really eat, 1000 green ones
>>>
```

If your program didn't work, go back and make sure everything is typed exactly as I typed it, and make sure you typed in the right windows. Before we go on to your first mini game I will explain more about the program you just made.

The Meaning of the Code

```
#My first program
```

This first line of your program was just for you. Remember that the computer ignores comments that begin with the # sign.

```
print( 'How many eggs can you eat?' )
```

This line was to tell the computer the words you wanted it to show or print on the computer screen.

```
amount=input()
```

The above like was to do two things. We made up the name of a variable and called it "amount." We also used the **input**() command to tell the computer that the computer user would put some data into amount variable/box.

```
print( 'Wow you can really eat,  ' +amount)
```

This line was to tell the computer to print these words on the computer screen but to add our variable. Our variable was called amount and now contains in memory whatever you typed for the input.

Change the lines above and make up your own mini programs to practice and see what happens. When you are ready, go on to the next part and make your first mini game.

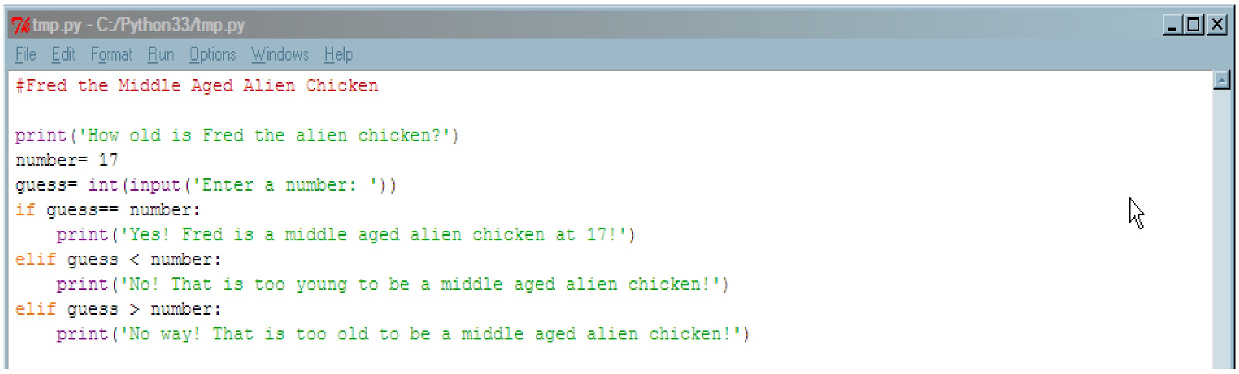
Conditional Statements

As a programmer you will often find yourself needing to use ***conditionals***, *(statements based on a condition)*. If you eat 2000 candy bars: you will get sick. This kind of conditional statement is called an "**if**" statement.

We will use the **if** and the **elif** statements to make your first simple game. Let's name your game; "Fred; The Middle Aged Alien Chicken."

To start programming I will make a note of the name of my game at the top of the window as a reminder, using the "#" key.

Enter the following code and then we can play and explain it:


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
#Fred the Middle Aged Alien Chicken

print('How old is Fred the alien chicken?')
number= 17
guess= int(input('Enter a number: '))
if guess== number:
    print('Yes! Fred is a middle aged alien chicken at 17!')
elif guess < number:
    print('No! That is too young to be a middle aged alien chicken!')
elif guess > number:
    print('No way! That is too old to be a middle aged alien chicken!')
```

Make sure you are in the right window and that you type everything exactly as you see above. Then click on *run* and *run module* to try it. Enter a number and it will give you a reply. Enter another number and ...ahh!! it just repeats what you entered! You close the game and run it again to try again. So, at this point, you can only make one guess before the game replies and stops working. Don't worry; we will fix that in a moment when you learn about loops. For now, I will explain the code we have so far:

```
#Fred The Middle Aged Chicken
```

Again the # sign makes this information to ignore for the computer, but a reminder or information for us humans and aliens who know how to read.

```
print ( 'How old is Fred the alien chicken?' )
```

This tells the computer to print the string of characters inside the ('') to the screen.

```
number=17
```

This tells the computer that the variable we called "number" will mean "17."

```
guess=int(input( ‘Enter a number: ’ ))
```

This tells the computer that the variable we call “guess” will mean; we want an integer input from the computer user. A string of characters, ‘Enter a number: ’ , to make the request inside the inner set of parenthesis. *What’s with the parenthesis in parenthesis?* This tells the computer what to do first. Just as in math, the parenthesis are done from the inner to the outer ones; the inner pair first, followed by the outer pair. Here, the string (‘Enter the number’) will be seen by the computer before the (input()) is expected. The inner parenthesis in (input()) has the meaning of (input (something here)). So, the computer displayed, “Enter a number” and waited for us to input an integer before going on.

```
 if guess==number:
```

This was to tell the computer that if the variable we called “guess” really is equal to the integer typed in the input : (then) do... The : sign means “then” in Python.

```
 print( ‘Yes!, Fred is a middle aged alien chicken at 17!’ )
```

This tells the computer to print or put the string inside the (‘’) on the computer screen.
Notice that this and all the print commands are indented the same amount. This will become important soon when we explain how to create blocks of code that tell the computer what goes together. We will do this to teach about loops and how to fix the little problem of our game only allowing one guess before it goes stupid and just repeats what we type again.

```
elif guess<number:
```

Here’s a new one. *What’s an elif?* No, not that. It means “or else if...” Here, we are telling the computer that in the past a guess was made and we told it what to do; but if what we said may happen, does not happen; here is something else to do. So, (elif), the “guess” variable, is less than the “number” variable we told the computer meant “17.” Read that again if you didn’t get it. Think of Python as an old man with the ability to think like Einstein but the slurred vocabulary of a drunken two year old. Not to criticize it; this is one of the ingenious aspects of Python. Saying little with a powerful meaning can help us make powerful programs by writing less code within a very short time.

```
 print( ‘No that is not too young to be a middle aged alien chicken’ )
```

Here we indent and tell the computer to print a string to the screen again.

```
elif guess>number:
```

This is a repeat of the elif above, but changing the meaning to; “if the guess is greater than the number then...”

```
print( ‘No way, that is too old to be a middle aged alien chicken’ )
```

Again, we indented and told the computer to print a string of characters.

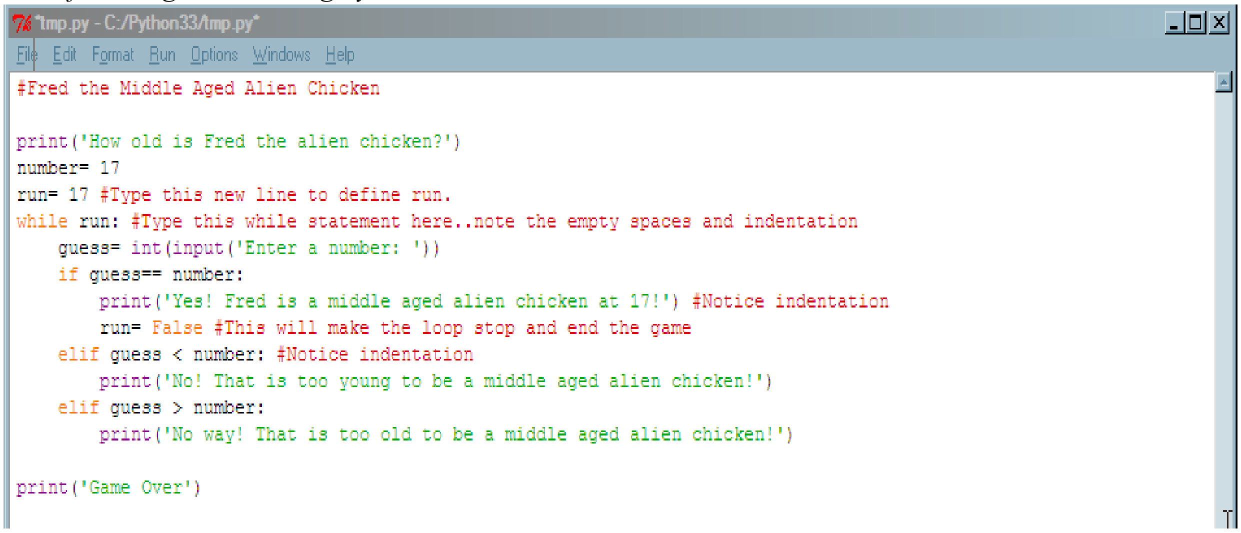
Remember pages ago when we talked about blocks? You were younger then, but maybe you remember. Well, we need to review that stuff and expand on the meaning to be able to teach you about loops. Only then will you know how to fix our games problem of stopping after one guess.

Loop For a “While” Statements

If you haven’t guessed it, *a **loop** is when the computer goes back and repeats something in a cycle until something breaks it out of that cycle.* The ***while statement*** *tells the computer to continue looping while some condition is continuing to happen; while it is still true.*

As long as some condition is true in a while statement, a block of statements or code will be repeated. It will break out of that loop when the condition is no longer true for that block of code.

Let’s fix our game. Change your code to look like this:


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
#Fred the Middle Aged Alien Chicken

print('How old is Fred the alien chicken?')
number= 17
run= 17 #Type this new line to define run.
while run: #Type this while statement here..note the empty spaces and indentation
    guess= int(input('Enter a number: '))
    if guess== number:
        print('Yes! Fred is a middle aged alien chicken at 17!') #Notice indentation
        run= False #This will make the loop stop and end the game
    elif guess < number: #Notice indentation
        print('No! That is too young to be a middle aged alien chicken!')
    elif guess > number:
        print('No way! That is too old to be a middle aged alien chicken!')

print('Game Over')
```

Most of this code was already explained, so we will just look at the changes and the new things. You didn’t need to type any of the red comments for your code to work. These were just there to help you. The spaces are also ignored by the computer and are placed in code to make it easier for humans to read. The indentations are important changes. We will discuss those in a moment. Now, if you try the game, you can continue guessing until you guess correctly. The game will only end when you guess the correct answer. Play around with the game and the code before moving on.

The code:

```
run=17
```

Here we named another variable called “run,” and told the computer that it also means “17.”

```
 while run:
```

The *while statement is known as a looping statement. It tells the computer to repeat the next block of code as long as it continues to be true.* Notice the empty line after the line of code with the while statement. In this example, I used a blank lines before and after the block of code that the while statement applies to. This entire block will continue to repeat forever until the run variable becomes false. So, the meaning of this would be to repeat the next block of code while the run variable is true; “then, (:), do this...” Here we can add if or elif statements to tell the computer when our run variable is true or when our run variable becomes false. If the run variable becomes false, our while run loop is no longer true and will stop looping/repeating.

```
    guess=int(input( ‘Enter a number: ’ ))
```

Here we created another variable called “guess.” We told the computer that when we use this word we mean the integer input that a person will type when he sees the string; (**‘Enter a number: ’**), on the computer screen.

```
    if guess==number:
```

This means that if the guess, (input), really equals, (==), the number, (17), then, (:)...

```
        print( ‘Yes! Fred is a middle aged...’ )
        run= False
```

These two statements are indented equally to tell the computer to run them if the guess ==number happens. This tells the computer that if this happens and the person inputs the number 17, then do the following indented things/lines of code. If this happens first print the string in parenthesis to the screen, then make the meaning of our run variable; “**False**.” This causes the while statement to now be false and the computer knows to stop repeating this loop and jump out of this set of indented lines, (or this block of code), and continue on the next block, (indicated by less indentation). That would make it jump to the print statement with the string (‘Game over’).

If the number 17 is not guessed, (our input), this would make the while statement continue to be true. So, the computer should continue and repeat the while block again by going to the next line in the block:

```
    elif guess < number:
```

This means, “or else if the guess is less than 17 then, (:)...

```
        print( ‘No, that is too young...’ )
```

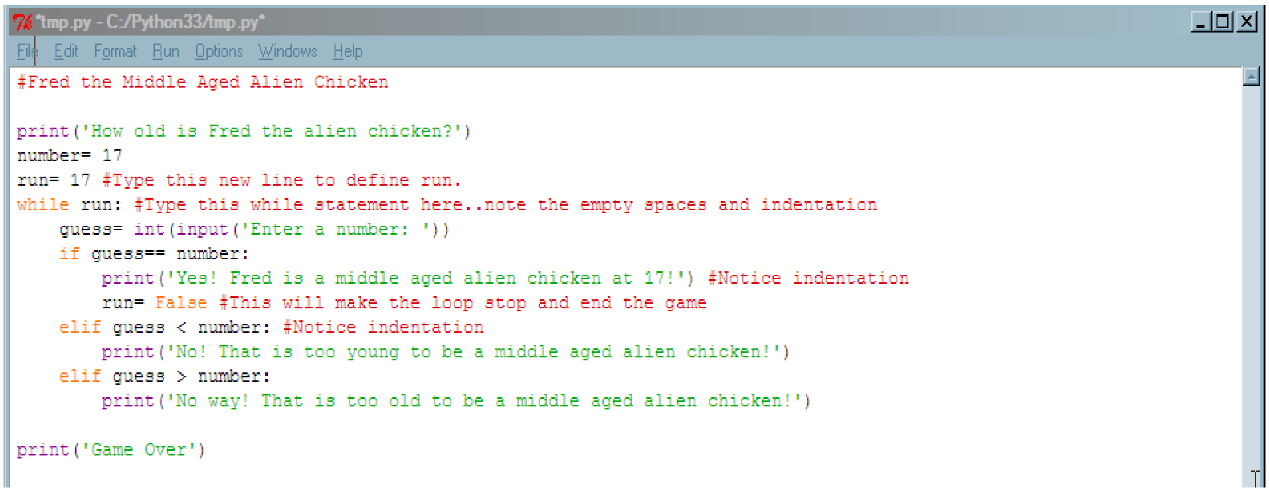
Print the string in parenthesis to the computer screen.

```
    elif guess > number:

# Or else if the guess is greater than 17 then...

        print( ‘No, that is too old...’ )

 # Print the string in parenthesis to the computer screen.
```

```
print( ‘Game over’ )
```

This statement will not run as long as the computer is looping the while block of code, because we let the computer know this is outside the while block by not indenting it. It will only run when our while block become false because 17 was the input.

Whew! Now you can understand how Python can save you time. Look at how many English words it just took to explain what we were doing. Then look again at how short the Python blocks of code are:


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
#Fred the Middle Aged Alien Chicken

print('How old is Fred the alien chicken?')
number= 17
run= 17 #Type this new line to define run.
while run: #Type this while statement here..note the empty spaces and indentation
    guess= int(input('Enter a number: '))
    if guess== number:
        print('Yes! Fred is a middle aged alien chicken at 17!') #Notice indentation
        run= False #This will make the loop stop and end the game
    elif guess < number: #Notice indentation
        print('No! That is too young to be a middle aged alien chicken!')
    elif guess > number:
        print('No way! That is too old to be a middle aged alien chicken!')

print('Game Over')
```

Getting Loopy (More on Loops)

Learning to think like a loop thinks is very important to understanding what will happen in the algorithms you write. Let's look at some simple while loop psychology to understand the looping process.

What will the following code do? *(Don't try it yet, just think about it.)*

```
x = 0
while x < 12:
   print(x)
```

**Remember that to type all of this in and run it you need to open a new window in IDLE. Don't try to use the Shell window because you will not be testing and running it line by line.*

If you didn't listen and actually ran this, you found that it will loop eternally and you have to kill it by closing the window. The better way to kill an eternal loop is to hold down the **control key and hit the "c"** key at the same time. Now that you know a better way to kill it, you can give it a try.

Why? Well, in the print function we told Python to print x, which is = to 0; and we told the loop to continue while x or 0 is less than 12. Zero will always be less than twelve so the loop will repeat forever until the user interferes to kill it.

Ok, now you know. *Try this one:*

```
x = 0
while x < 12:
      print(x)
      x = x + 3
```

Before running it, what do you think will happen?

(Don't move on to the next page until you have a theory.)

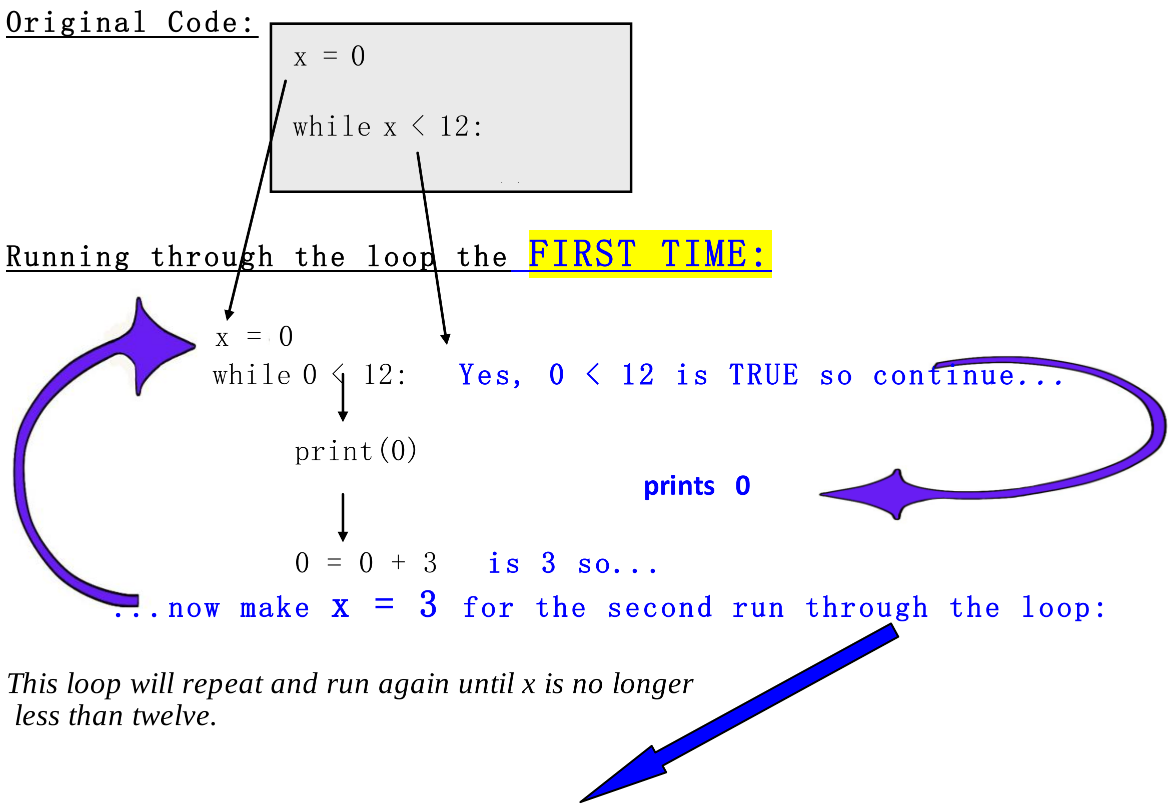
Your output should be:

```
0
3
6
9
```

So, what exactly is happening here?

```
x = 0 # we defined x as being 0
 while x < 12: # we told Python to loop as long as x is less than 12
      print(x) # we told Python to print x every loop as long as the above
                  conditions are being met.
        x = x + 3
```

This last statement is redefining what x is every loop. The first loop it prints 0 then goes to this x = x +3 statement so now: x = 3 for the next loop.



Original Code:

```
x = 0

while x < 12:
```

Running through the loop the **FIRST TIME:**

```
x = 0
while 0 < 12:    Yes, 0 < 12 is TRUE so continue...

      print(0)
                                  prints 0

      0 = 0 + 3    is 3 so...
...now make x = 3 for the second run through the loop:
```

This loop will repeat and run again until x is no longer less than twelve.

Running through the loop the **SECOND TIME:**

Running through the loop the **SECOND TIME:**

```
x = 3
while 3 < 12:    Yes, 3 < 12 is TRUE so continue...

      print(3)
                               prints 3

      3 = 3 + 3   is 6 so...
...now make x = 6 for the second run through the loop:
```

Running through the loop the **THIRD TIME:**

```
x = 6
while 6 < 12:    Yes, 6 < 12 is TRUE so continue...

      print(6)
                               prints 6

      6 = 6 + 3   is 9 so...
...now make x = 9 for the second run through the loop:
```

Running through the loop the **FOURTH TIME:**

```
x = 9
while 9 < 12:    Yes, 9 < 12 is TRUE so continue...

      print(9)
                               prints 9

      9 = 9 + 3   is 12 so...
...now make x = 12 for the second run through the loop:
```

Running through the loop the **FIFTH and FINAL TIME:**

Running through the loop the FIFTH and FINAL TIME:

```
x = 12
while 12< 12:   FALSE! 12 IS NOT < 12 SO BREAK LOOP!!!
```

Wow! If we looped that much we would be dizzy. Fortunately, the computer can handle it and all that looping just results in a simple output for us humans:

```
0
3
6
9
```

There are different ways to exit a while loop. One way is to just add the word "***break***" and the loop will end at that point. Another common way to end a loop is to define when it is true and when it is false.

```
while done = = false:
    for...(here I would enter my code block)
           if...(this will repeat in the loop until it becomes true)
                done = true (when done is true, we exit the loop, and are done with it)
```

Are you ready for a quiz? If not; go back and loop until you go it; if you think so; go on to the next page.

Review

Quiz 3:

Answer the following questions on a separate piece of paper or in a document if you prefer. Check your answers at the end of this lesson.

3.1 Define the following:

1. If statement
2. Loop
3. While statement

3.2 *Answer the following questions*:

1. Give two examples of conditional statements.

2. *Correct the following portion of code (without referring to the text):*

```
number = 17
run = 17
while run:
guess = int(input(Enter a number:)
if guess = number:
print  'Yes.'
run = False
elif guess < number
```

CHALLENGE:

Write a while loop that does a countdown from ten to one and then prints, "Go!"
*Check the end of this lesson to see one possible solution. *Hint: The "break" statement can be used to break out of a loop.*

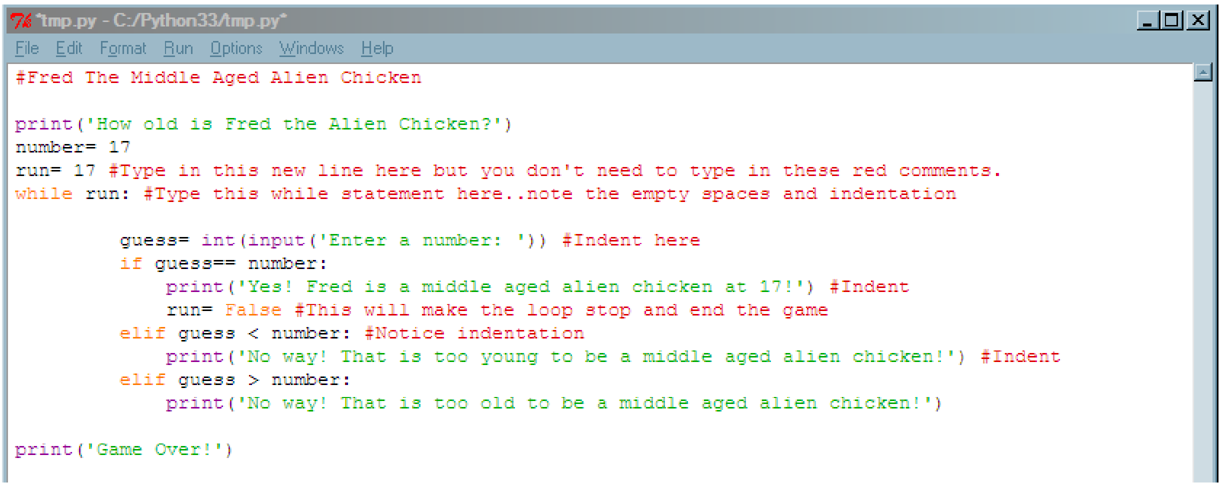
Vocabulary

If Statement *a conditional statement/ a statement based on a condition.*
Loop *when the computer goes back and repeats something in a cycle until something breaks it out of that cycle.*
While statement *tells the computer to continue looping while some condition is continuing to happen; usually while it is still true.*

3.2 *Answers*:

1. "if" and "elif," (you could also include "else").
2. *Obsessive attention to details and syntax must be developed as early as possible to be a great programmer. It is better to spend time developing these skills than in debugging messed up code after you write it. Prevention is better than repair but both skills need to be developed.*

Code:	*Errors:*
`number = 17`	
`run = 17`	
`while run:`	
`guess = int(input(Enter a number:)`	***indent, 'quotation marks', parenthesis))***
`if guess = number:`	***indent***
`print 'Yes.'`	***indent, quotation marks around 'Yes'***
`run = False`	
`elif guess < number`	***indent, missing colon : after "number"***


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
#Fred The Middle Aged Alien Chicken

print('How old is Fred the Alien Chicken?')
number= 17
run= 17 #Type in this new line here but you don't need to type in these red comments.
while run: #Type this while statement here..note the empty spaces and indentation

         guess= int(input('Enter a number: ')) #Indent here
         if guess== number:
             print('Yes! Fred is a middle aged alien chicken at 17!') #Indent
             run= False #This will make the loop stop and end the game
         elif guess < number: #Notice indentation
             print('No way! That is too young to be a middle aged alien chicken!') #Indent
         elif guess > number:
             print('No way! That is too old to be a middle aged alien chicken!')

print('Game Over!')
```

Challenge *(one possible answer, (you figure out the correct indentation)):*

`x = 10     while x <=10:   print(x)   if x < 1:   print("Go!")   break   x = x-1`

Lesson 4: Basic Python Parts

<u>What You Will Learn</u>:

- **Lists**
- **Slices**
- **Tuples**

Lists

*A **list** is a sequence of elements.* A list is referred to as an "*array*" in some programming languages. You could make a list of the family members in your home:

```
Family = ['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Cat', 'Cockroaches']
```

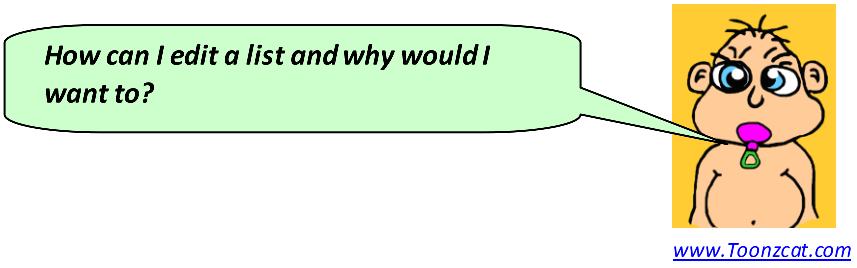
Each element in the list has a position of reference. In other words, they are numbered for indexing or referring to them later. *The elements within the brackets are often called the "**index.**"* But, instead of calling the first element in the list; "1," programmers begin with "0." So, now you won't be confused if you see a programmer counting how many cups of coffee he had tonight by beginning with; "0, 1, 2, 3..." Lists can store numbers, strings, both, and they can store other lists.

They may look like this:

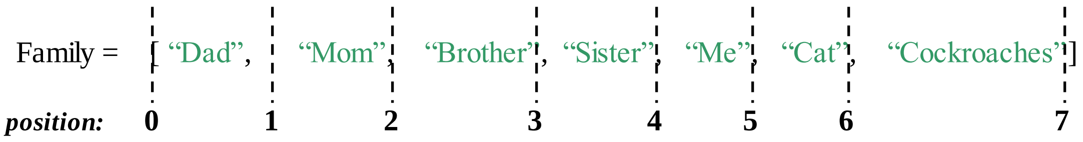
```
Numbers:      theAlist=[ 2, 4, 6, 8 ]
Strings:       theBlist=['words', 'abc', 'fuglet']
Both:          theClist=['abc', 4, 'words', 7]
Other Lists:   AllLists=[theAlist, theBlist, theClist]
```

We can edit lists. But, the word "*edit*" is too common for a programmer to use, so let's use the word "*mutable*."

Later you will learn about tuples. Tuples are immutable. Huh? ***Mutable*** *means we can edit it.* ***Immutable*** *means we can't edit it.* So the secret meaning is that lists can be edited; but strings and tuples can't be edited, (*directly*).



Maybe you would like to edit your list of family members. Remember our list:


Family = ["Dad", "Mom", "Brother", "Sister", "Me", "Cat", "Cockroaches"]
Index position: 0 1 2 3 4 5 6 7

Let's say you bought a mouse and he ate the family cockroaches. It was a sad event but life goes on and now you must edit your list of family members. You need to replace "Cockroaches" with "psycho mouse."

To replace an item in a list you need to understand what is in a particular index position. Index position references begin before an item and at the comma after the item in the list. Since "Cockraoches" is in the 6th position of our "Family" list, (remember you count from 0), we do this:

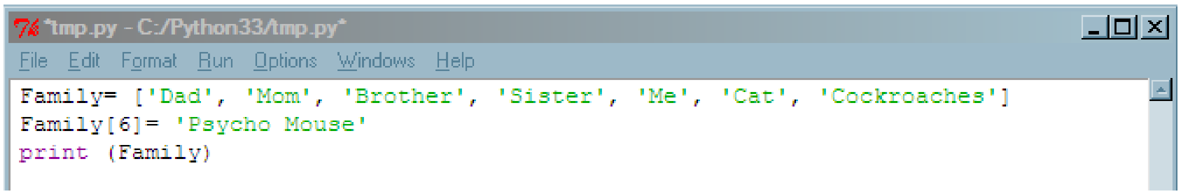
Family[6]= "Psycho Mouse"

Give it a try. First, define what is in your list by typing:
Family = ["Dad", "Mom", "Brother", "Sister", "Me", "Cat", "Cockroaches"]

Then, type:
Family[6]= "Psycho Mouse"

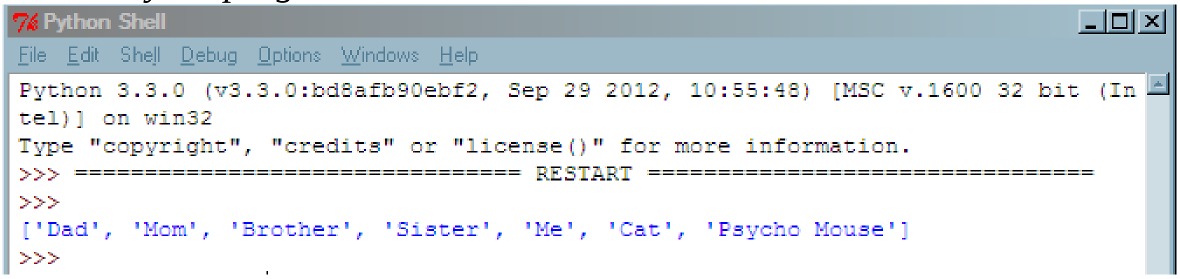
Test it by typing:

print(Family)


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
Family= ['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Cat', 'Cockroaches']
Family[6]= 'Psycho Mouse'
print (Family)
```

Now run your program.

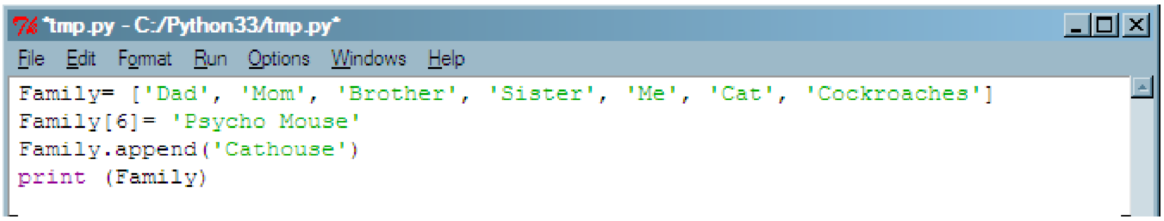


```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Cat', 'Psycho Mouse']
>>>
```

What happens if I want to add something to my list? *Adding something to a list is referred to as **appending** the list.*

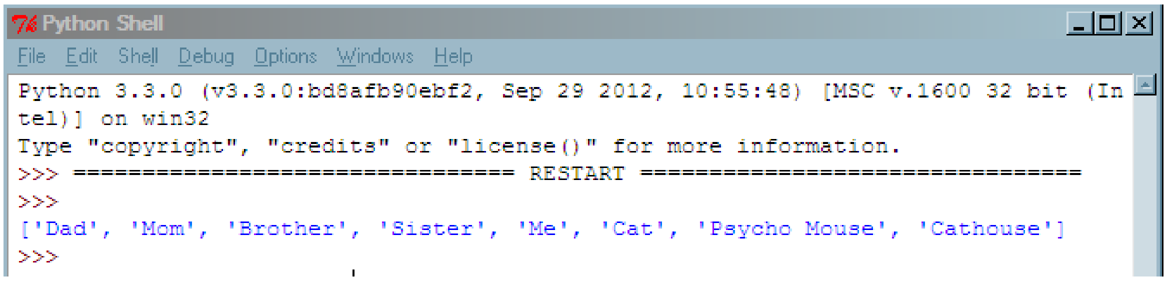
Let's say that our Cat and Psycho Mouse got married and had a baby. We will call it, "Cathouse." Now we have a new member of our family and we must append the list.
We add this line before our print statement:

```
Family.append( "Catouse" )
```


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
Family= ['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Cat', 'Cockroaches']
Family[6]= 'Psycho Mouse'
Family.append('Cathouse')
print (Family)
```

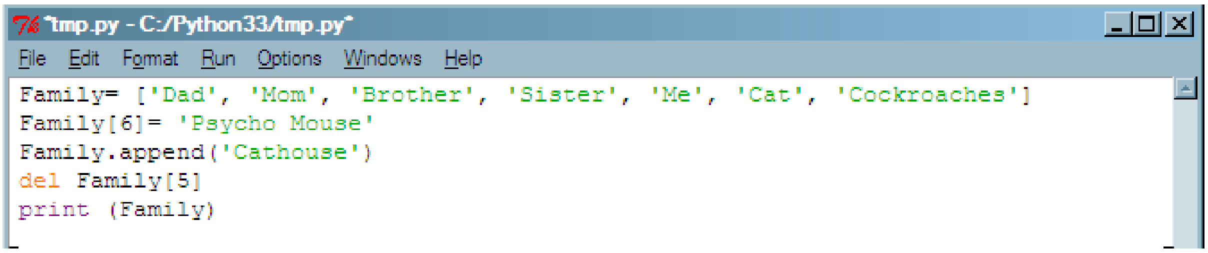
Run it:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Cat', 'Psycho Mouse', 'Cathouse']
>>>
```

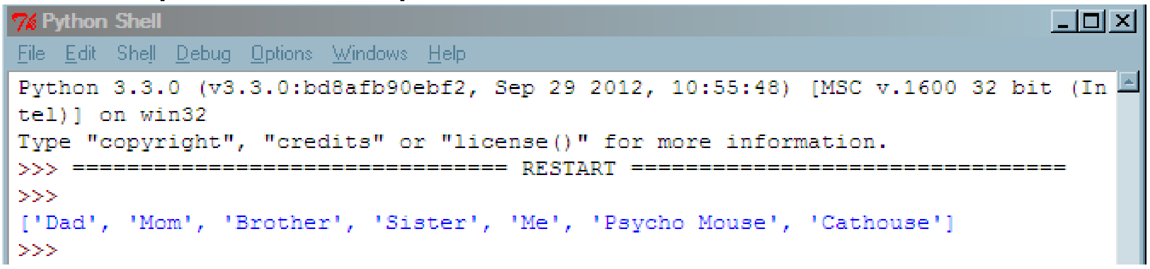
How do I remove an item? I'm glad you asked. Let's say that Psycho Mouse gets angry and eats the Cat. To remove "Cat" from the Family list we use **del** for delete and give the index position:

```
del Family[5]
```


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
Family= ['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Cat', 'Cockroaches']
Family[6]= 'Psycho Mouse'
Family.append('Cathouse')
del Family[5]
print (Family)
```

Run it to confirm the removal of the cat:



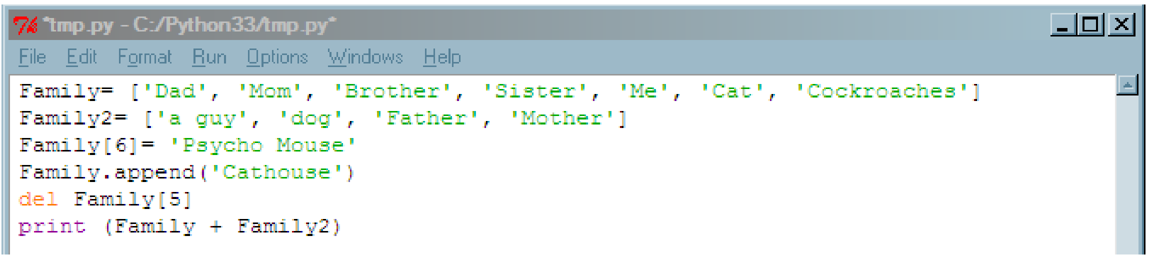
```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Psycho Mouse', 'Cathouse']
>>>
```

Just to complicate our lives, our sister marries a guy and now another family has become part of our family. Fortunately, adding two lists together is easier than combining two families. First we define what is in the second list. Let’s add this list second family list under our Family list:

```
Family2=[ “a guy” ,  “dog” ,  “father” ,  “mother” ]
```

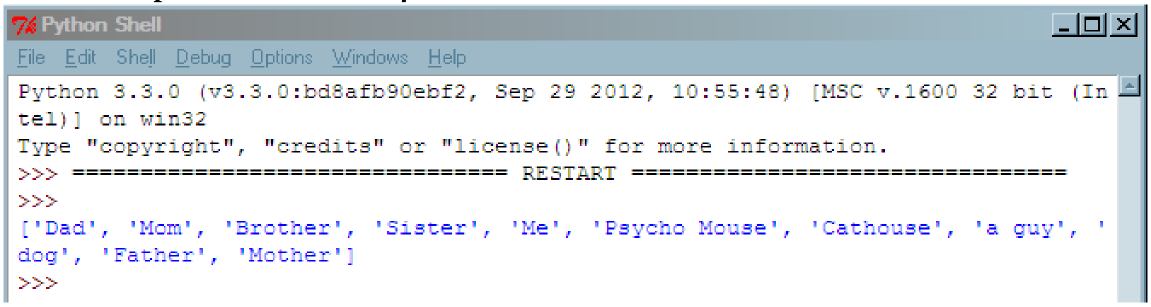
Then to add the two lists change the print statement to:

```
print(Family+Family2)
```


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
Family= ['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Cat', 'Cockroaches']
Family2= ['a guy', 'dog', 'Father', 'Mother']
Family[6]= 'Psycho Mouse'
Family.append('Cathouse')
del Family[5]
print (Family + Family2)
```

Run it to output a combination of the two lists:



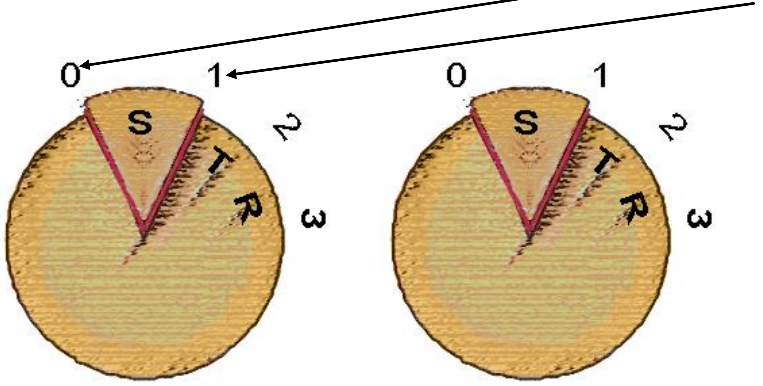
```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
['Dad', 'Mom', 'Brother', 'Sister', 'Me', 'Psycho Mouse', 'Cathouse', 'a guy', '
dog', 'Father', 'Mother']
>>>
```

Slices

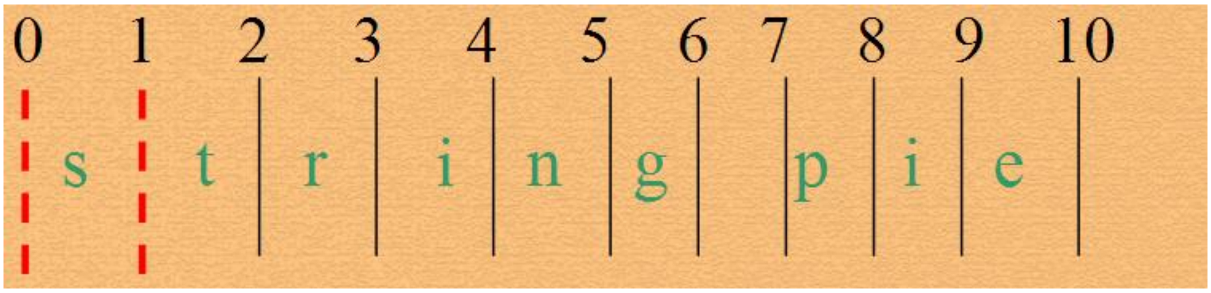
Remember how we used an index to cut one part of a list out to exchange it or delete it?
Wouldn't it be great if you could choose any part of the list or a string; one character; half of an index; all of the list; or any part you want to copy or work with? You can! *A slice is cutting a list into smaller parts. A **slice** is a subset of a list or a string.* Just as you can choose what piece of pie you want to cut and how big you want to cut it; you can cut a slice from a list or a string starting and ending where you want it.

Let's cut up a string pie. Don't worry; you don't need to eat it. The strings always get caught between your teeth anyway, (but they are good fiber). Remember that we use a 0 for our first index position. But, when talking about slices of pie or strings you have to use two reference points. You need an index or reference point for the first cut and one for the final cut of the slice. So, our index points are on each side of the characters in our string.

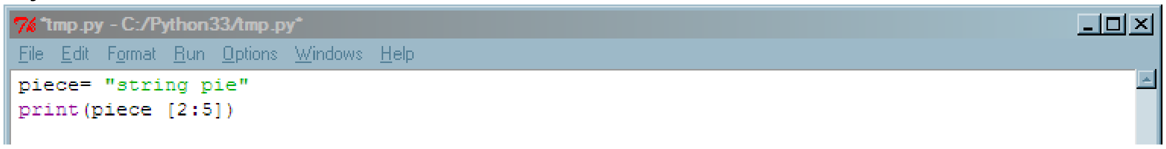
If we want to slice out the "**s**" we have to tell the computer the ***beginning and ending index for the slice, [0:1].***



It is probably easier to think about it this way:

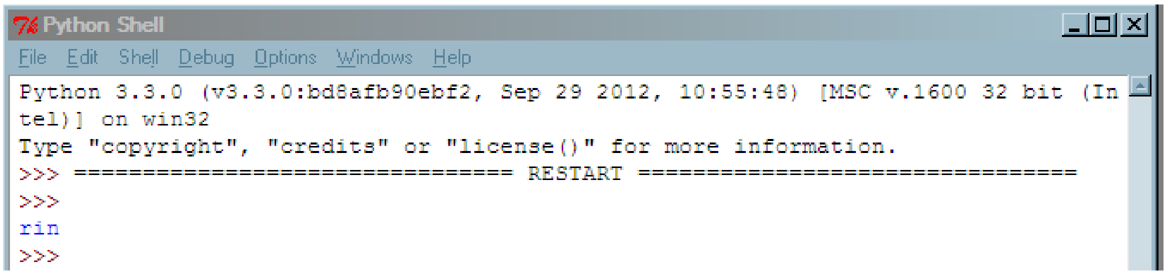


Try this:


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

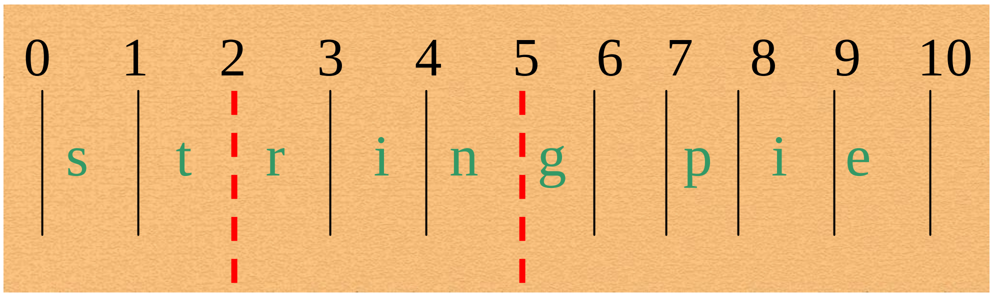
```
piece= "string pie"
print(piece [2:5])
```

Run it:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
rin
>>>
```

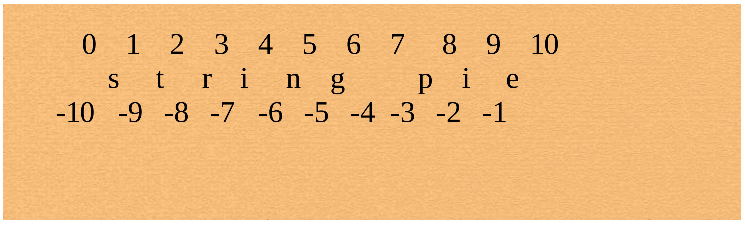
I'm not sure what a "rin" tastes like but we successfully sliced a piece of our string from index position 2 to 5!



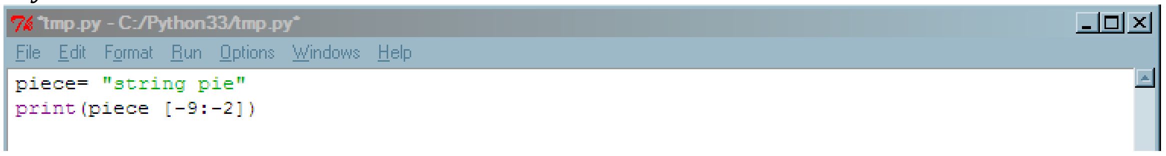
You try a few to practice. First choose the letters you want to cut out of the string. Then choose the beginning and ending cut for your slice of the string. Practice until you are confident in choosing the index positions and getting the slice you want.

There is another way to refer to index positions in a slice if you want to use it. You can think backwards! There are practical uses for this in more advanced programming.

For now, I will only introduce you to it and you can play with it. In this index system, index positions use negative numbers.

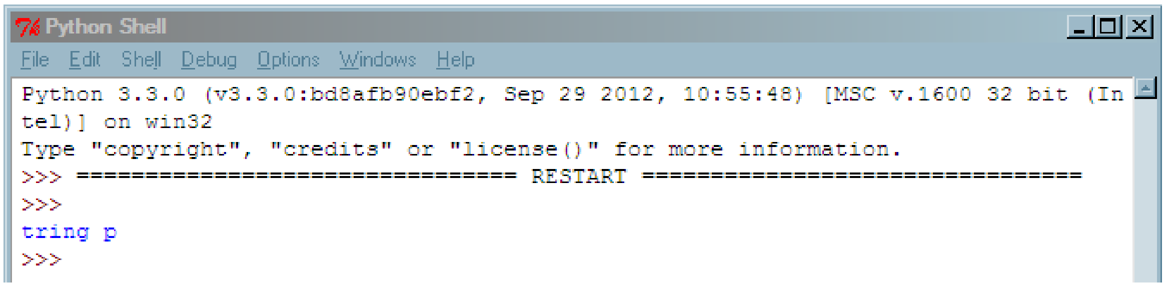


Try it:


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
piece= "string pie"
print(piece [-9:-2])
```

Run it:



```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:55:48) [MSC v.1600 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
tring p
>>>
```

A "tring p" probably tastes better than a "rin," but I'm only guessing. When you program you will find that making a copy of part or all of what you have typed before will save you a lot of time. Slicing strings is a useful talent.

<u>Practice 4.1   Do this</u>:

Look at what we have done so far, and take a guess about what you think will happen before doing the following:

1. What do you think happens if you create a string and you print [:3] of that string?
2. What happens if you create a string and you print [3:] of that string?

Come on! Take a guess. Try it before looking at the answer at the end of the lesson.. Learning by doing it and thinking about it makes things easier to remember.

Here are a couple more slicing operations you can play with:

```
>>> string1='pronounce'

>>> string2=' deepfat'

>>> newstring = string1 + string2

>>> print(newstring)

pronouncedeepfat
```

In the previous example you can see how easy it is to create a new string variable that contains the combined data of other variables.

```
>>> newstring = string1[:3] + string2[4:]

>>> print(newstring)
```

If you put a space here [4:], rather than [4:], it will not work.

In the above example you can see how easy it is to create a new string that is a combination of parts of other strings. These same slice operations work for lists. *What is the difference when slicing lists or strings?* With strings each character and empty space is indexed as separate data. Lists are *compound data types* where each chunk of data is indexed according to the location of the commas you used to designate each item of data. In other words, with strings you would count index positions before and after each character or empty space; with lists you would count index positions before the first item as 0 and at each comma after each item in the list.

<u>String index positions</u>:

```
     s    t    r    i    n    g
0    1    2    3    4    5    6
```

<u>List index positions</u>:

```
Mylist = ['pickles', 'catchup', 'hamburger', 'bread',  '20', 4, 1000]
       0          1            2            3         4    5  6      7
```

Strings and lists also differ in the fact that strings are ***immutable***, *(cannot directly edit them)*, while you can change and edit items in lists. *Try this:*

```
>>> list1 = ['up', 'down', 300, '40']

>>> list1[2]=list1[2]+77

>>> print(list1)

['up', 'down', 377, '40']
```

But what happens if you add something to an item that is not an integer?

```
>>> list1[1]=list1[1]+'erk'

>>> print(list1)

['up', 'downerk', 377, '40']
```

You can also ***append***, (add something to the end), a list:

```
>>> mylist = ['up', 'downerk', 377, '40']

>>> mylist.append('kitten')

>>> print(mylist)

['up', 'downerk', 377, '40', 'kitten']
```

What else can you do to edit lists? I'm glad you asked!

You can:

Replace items in a list:

```
>>> mylist = ['box','pen', 4, '100', 'hat', 'horse']

>>> mylist[0:3] = ['rug', 2, 'pencil']

>>> print(mylist)

['rug', 2, 'pencil', '100', 'hat', 'horse']
```

Remove items:

```
>>> mylist[0:2] = []

>>> print(mylist)

['pencil', '100', 'hat', 'horse']
```

Insert items:

```
>>> mylist[3:3] = ['onions', 'carrot', 44]

>>> print(mylist)
```

Insert a copy of itself:

```
>>> mylist [:0] = mylist

>>> print(mylist)

['pencil', '100', 'hat', 'onions', 'carrot', 44, 'horse', 'pencil', '100', 'hat',
'onions', 'carrot', 44, 'horse']
```

Or in a copy to another location within the list:

```
>>> mylist [2:2] = mylist

>>> print(mylist)

['pencil', '100', 'pencil', '100', 'hat', 'onions', 'carrot', 44, 'horse',
'pencil', '100', 'hat', 'onions', 'carrot', 44, 'horse', 'hat', 'onions',
'carrot', 44, 'horse', 'pencil', '100', 'hat', 'onions', 'carrot', 44, 'horse']
```

Let's find out how many items are in our list:

```
>>> len(mylist)
```

Let's clear the list:

```
>>> mylist[:] = []

>>> print(mylist)

[]
```

You can now understand why lists will be one of your most valuable tools for manipulation of data. Play and practice with these until you are confident.

Tuples

Let's talk tuples! *A **tuple** is like a list but can't be edited or changed, (immutable), in its original form.* If we really want to be able to modify tuple, we must convert it to a list by using the list(tuplename) first. We generally use parenthesis, ( ), rather than brackets, [ ], to tell Python we are defining a tuple. If it is very obvious to Python that it is a tuple, parenthesis can be skipped.

This is a *string*:

```
Family3 =  'creatures'
```

This is a *tuple*:

```
Family3 = ( 'a guy' ,  'dog' ,  'cat' )
```

This is a *list*:

```
Family3 = [ 'a guy' ,  'dog' ,  'cat' ]
```

Tuples are not easily changed so why would we use them?

As we mentioned; a program can run a tuple with more efficiency than a list. Unless the program is very small, tuples are faster. So, if you don't need to change your list data, it's generally better to use a tuple. Tuples can contain many *things you may not want to change or "read only" code.* These ***constants*** in your program may be; music, a sequence of sound files, a sequence of images, etc. You can assign an individual item to a variable or you could assign a group of items in a sequence to a tuple. Tuples can also be ***nested***, *(placed), inside* of other tuples; called up as *functions*; and can hold any data type. For some applications, tuples are used much more often than lists. We will be getting into functions later. For now, just understand the basic principles and the importance of tuples and lists.

Manipulating data with strings, lists, and tuples is going to be a major part of your daily routine as a programmer. It would be best for you to think of your own scenarios where you might need these skills and practice more before going on.

If that sounds boring then continue on, don't let me stop you. You can always return to this lesson if you feel you missed something. You will be using these skills continually from now on, so you will eventually master them in the context of your work.

So, what are some the other differences between tuples lists and strings? Check out what methods are available for each of these and get more information by typing: dir(list), dir(string), help(list), help(string). You will understand more about what you see when you study methods in future lessons. For now, just remember how to find more help and information within Python on these very important data types.

Review

Quiz 4:

Answer the following questions on a separate piece of paper or in a document if you prefer. Check your answers on the next page.

4.1 Define the following terms:

1. Immutable
2. List
3. Mutable
4. Nested
5. Slice
6. Tuple
7. constants

4.2 Questions:

1. What is the most important difference between a tuple and a list?
2. If “mooplug” is a string, what is at index position 0?
3. How do we modify a tuple?
4. How do we remove an item from a list?
5. Show an example of appending a list called, “girls.”

4.3 Activity:

1. Practice making lists, appending them, and removing items until you are confident with indexing references.
2. Make some tuples and convert them so you can edit them.

VOCABULARY

***Immutable** means we can't edit it.*
***List** a sequence of elements*
***Constants** are read only data*
***Mutable** means we can edit it.*
***Nested**, (placed), inside*
***Slice** a subset of a list*
***Tuple** like a list but can't be edited or changed in its original form.*

4.2 Answers:

1. Tuples are immutable while lists are mutable.
2. "m" is in index position 0.
3. Convert it to a list?
4. Use "**del** Listname[index position]".
5. girls.append(**"girl2"**)

4.3 Activity

Answers will vary. Refer back to the lesson examples if you don't fully understand.

Practice 4.1 Answers:

1. [:3] produces everything in your list up to index position 3.
2. [3:] produces everything in your list after index position 3.

Remember to type the syntax correctly.

```
>>> mystring = 'astring'
>>> print(mystring [3:])
ring
>>> print(mystring [:3])
ast
```

Lesson 5: Functions

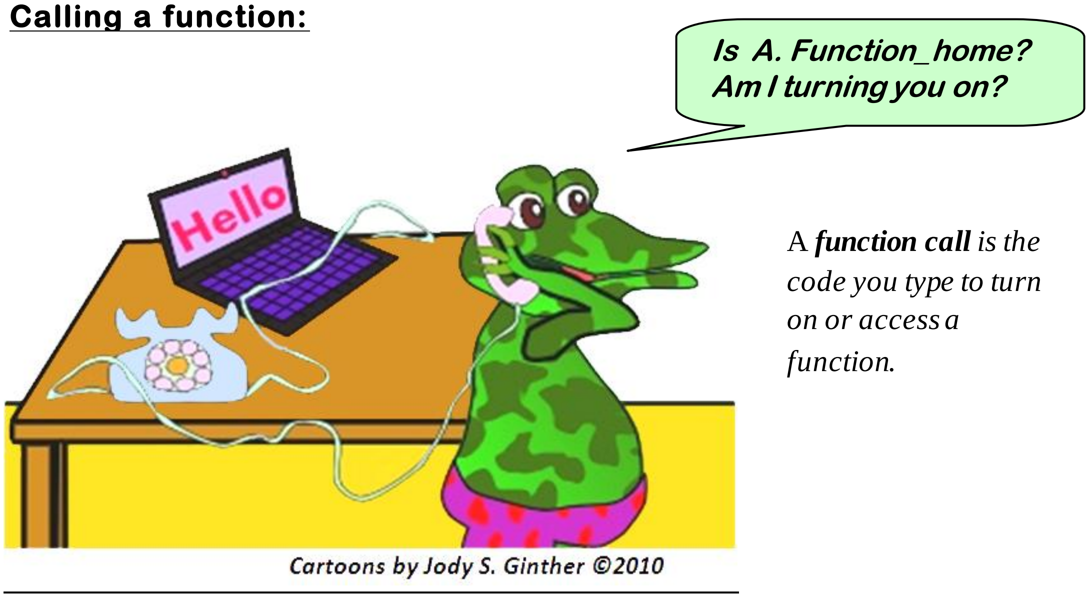
What You Will Learn:

- What's a Function
- Making Your Own Functions
- Arguments And Parameters
- Docstrings and Suites
- global and Local scope

What's a function?

*Again; a **function** is a re-usable mini program inside of your program.* One of the greatest things about functions is that you never have to repeatedly type code, ***if you need to repeat it, let a function do it.*** You have already been using functions throughout this book. The "`print()`" *function call* for the print function is the one you have used most.

Calling a function:



A ***function call*** *is the code you type to turn on or access a function.*

MAKING YOUR OWN FUNCTIONS?

It is common in writing code to use the same code many times. If we assign this code to a function, we only need to call that function again rather than retype the code. To define the meaning of a function for the computer we use the **def** statement.

The def statement *is made up of:*

the keyword, def

*our function **name***

*followed by **parenthesis** where you can add parameters or arguments*

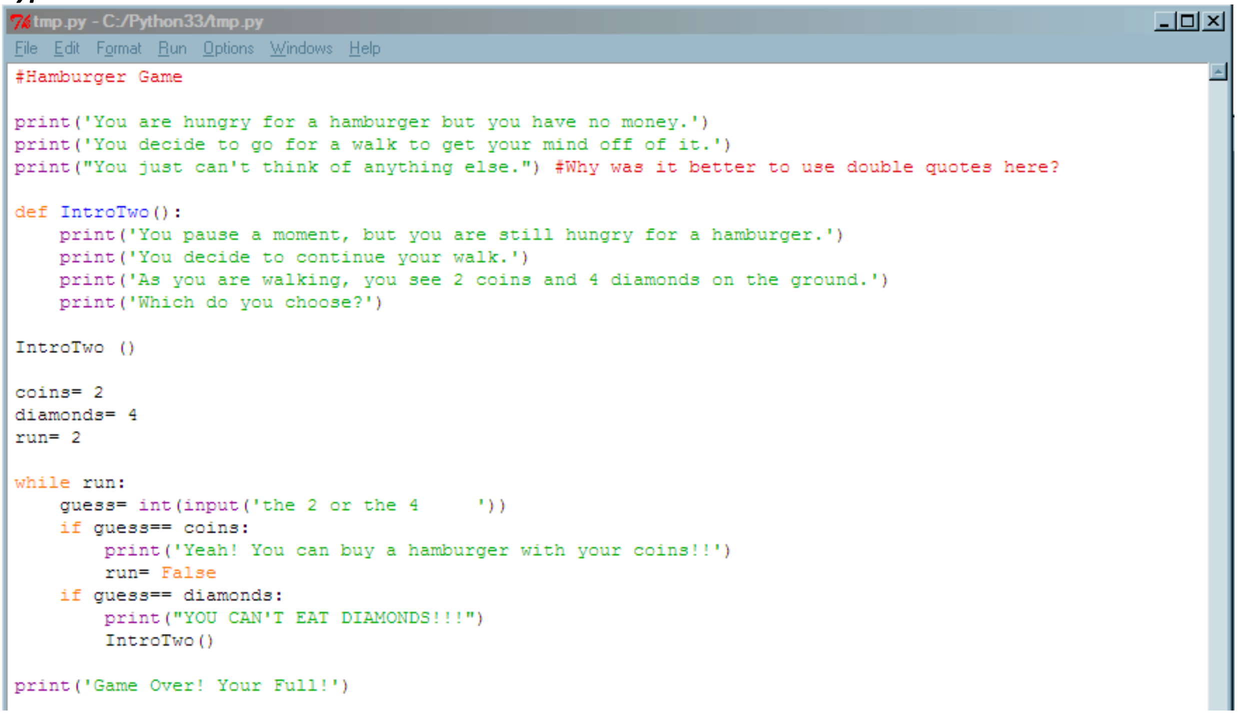
*a **colon***

*and finally the **def-block**.*

```
def  thename  ():
   Here we put the details of our definition block.
   We can put other functions or code here.
```

Let's try it. Below is the code for a new game based on our Alien Chicken game. The purpose of this game is to show you that by making and using your own functions, you can save a lot of time rather than retyping things. We have a two part introduction to this game. The second part will be repeated until the game is over. So, this part will be defined as a function and called up when we need it.

Type this:


tmp.py - C:/Python33/tmp.py
File Edit Format Run Options Windows Help

```
#Hamburger Game

print('You are hungry for a hamburger but you have no money.')
print('You decide to go for a walk to get your mind off of it.')
print("You just can't think of anything else.") #Why was it better to use double quotes here?

def IntroTwo():
    print('You pause a moment, but you are still hungry for a hamburger.')
    print('You decide to continue your walk.')
    print('As you are walking, you see 2 coins and 4 diamonds on the ground.')
    print('Which do you choose?')

IntroTwo ()

coins= 2
diamonds= 4
run= 2

while run:
    guess= int(input('the 2 or the 4     '))
    if guess== coins:
        print('Yeah! You can buy a hamburger with your coins!!')
        run= False
    if guess== diamonds:
        print("YOU CAN'T EAT DIAMONDS!!!")
        IntroTwo()

print('Game Over! Your Full!')
```

I hope you enjoyed this free mini-book. The full course can be found by following the link at www.Toonzcat.com, www.Quantum-Sight.com, or www.QuantumSight.mobi

Index F: Files Included with This Course

You can download the working files, (FilesAndAssets.rar), for this course at www.toonzcat.com

star_field.py
Space_cruise.py
plasmaball.png
Eyes.gif
explosion.gif
spaceship.png
setup.py
Asteroid_Fight.py
Alpha.py

For data folder:
asteroid.png
boom.wav
explosion.gif
laser.wav
music.wav
plasmaball.png
spaceship.png
Asteroid_Fight.ico