

INFORMATICS

Lecture 1

I210 – INTRODUCTION TO PROGRAMMING WITH PYTHON

Fill out this survey:
<https://goo.gl/forms/E2dk6NbKiXwUi7I22>

Today

- **Introductions**
- **Course Policies**
- **Introduction to:**
 - **Python**
 - **programming**
 - **algorithms**

If you miss anything today, please note that all slides from lecture are posted to Canvas on Tuesday and Thursday after the last lecture.

Your Instructors

John Duncan (I go by “J”)



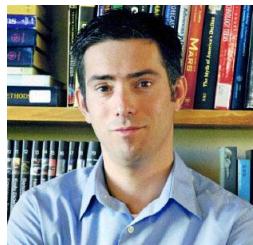
- PhD, Indiana University
- Areas: Education, Security, Privacy, Social Informatics, Pervasive Computing, AI, Cognitive Science

- Email: johfdunc@indiana.edu
- Office location: Undergraduate Annex 201
a.k.a. Ravenclaw House
- Office hours: Tuesday 1 – 3 PM or
By Appointment

Facilitator, Nothing Is Binary – a SoIC community for students, faculty, and staff who identify as GLBTQ or allies. Talk to me if interested!

Your Instructors

Johan Bollen



- PhD, University of Brussels
- Research Areas: Complex networks and systems, Data science, Web science

- Email: jbollen@indiana.edu
- Office location: Info East 305
- Office hours:
Tuesday 3 – 5PM or By Appointment

Your Instructors

Daniel Pierz



- MA, Indiana University
- Areas: Education, Information Science, Human Computer Interaction , User Oriented Design, Interactive Systems, Web Accessibility

- Email: dpierz@umail.indiana.edu
- Office location: Undergraduate Annex 202
a.k.a. Ravenclaw House
- Office hours:
MW 11AM - Noon or By Appointment

Your Instructors

Erika Lee



- MA, Indiana University
- Areas: Digital Design, HCI, Web & Interactive Development, User Oriented Design, Web Accessibility, Data Journalism

- Email: ebigalee@indiana.edu
- Office location: 611 N Woodlawn, Rm 101A
a.k.a. Gryffindor House
- Office hours:
Monday 9 – 11 AM or By Appointment

Why learn to program?

Programming lets us control computers

Making them do whatever we want, even if it's something no one has ever done before

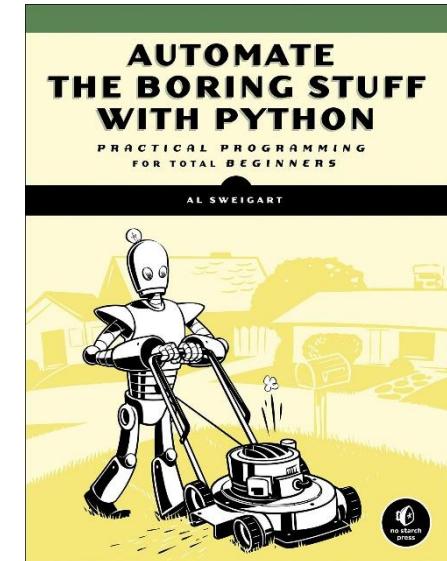
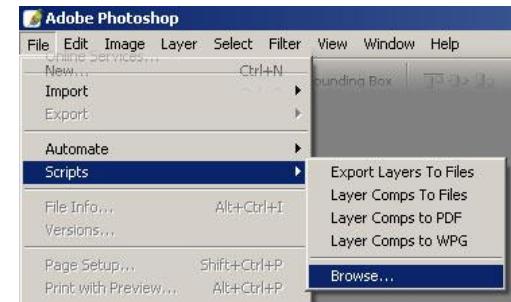
Learning to program will help you:

- Create webpages, software, apps, games, etc.
- Analyze research data, optimize business strategies, etc.
- Understand how computers work, and their capabilities and limitations
- Be successful! Have fun! Get a job!

But I Don't Want to Be a Programmer!

Even non-programmers benefit!

- Common tools in design often have programming-like scripting elements for better control
- Automating large and repetitive tasks is a huge time-saver
- Your job may involve supervising or cooperating with colleagues who program.
- *Wouldn't you like to be able to meaningfully discuss technology?*



Programming involves...

1. Breaking down a problem into concrete steps that a computer can perform
2. Writing these steps in a language the computer can understand
3. Resolving any issues caused by your solution

We'll learn how to do this process in I210

Key to learning: lots of practice!

We'll have in-class activities, labs, group projects, assignments....

My Background with Programming

- Time
- Languages
- Professional Experience

***Programming is a LEARNED SKILL,
not an in-born talent.***

Learning Languages

Treat learning to program like learning a foreign language...

You have to practice using it as much as possible!



AIs and UIs

Graduate (AIs) and Undergraduate (UIs) student assistants.

- Introductions
- Contact info on Canvas.
- *If you do well, you could be a UI also.*



*Your instructors,
AIs, UIs - we're
here to help!*

Peer Led Team Learning

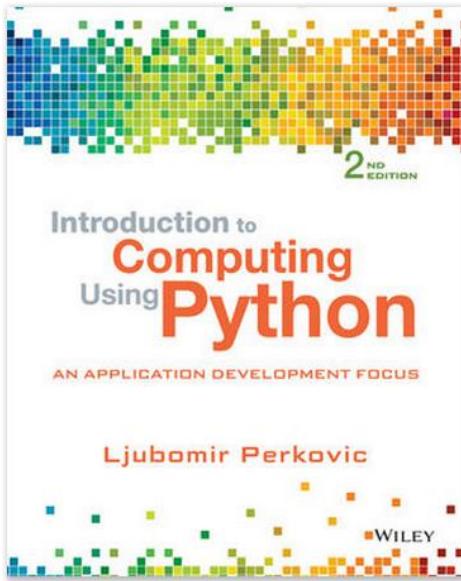
Peer Led Team Learning (PLTL)

1 PLTL Leader per class

- Extra credit will be awarded for attending PLTL sessions. *Earn 2 lecture work points for each session attended – Max 20 bonus points.*
- Research shows significant improvement in grades for participants, so try it!
- Not for:
 - HW help
 - Last minute exam prep



Course Materials



Intro to Computing Using Python,
by Ljubomir Perkovic (2nd edition)

- Python 3

Rent or buy here:

<http://www.amazon.com/Introduction-Computing-Using-Python-Application/dp/1118890949>

- Slides, Syllabus, etc. on Canvas
- You will submit your assignments on Canvas

Grading

35%	<p>Lecture Work (20%)</p> <ul style="list-style-type: none">– In teams, 1 submission – show it to us. <p>Lab Tasks (15%)</p> <ul style="list-style-type: none">– In teams, 1 submission – show it to us.
65%	<p>Retrospective Quizzes (20%)</p> <ul style="list-style-type: none">– Done out of class on Canvas, open course materials. <p>Individual Homework (20%)</p> <ul style="list-style-type: none">– Drop lowest 1, done <i>individually!</i> <p>Individual Lab Practicals (35%)</p> <ul style="list-style-type: none">– Taken during lab, open course materials.

Academic Misconduct

- **You may *not* get help from non-instructors or share code on the Homework or the Lab Practicals.**
 - Submissions must be YOUR work only.
 - DO NOT show/share your individual work with others. This includes asking others to debug your code.
- Don't submit code from the internet as your work.
 - *If you can find it, or buy it, so can we!*
- The standard penalty for any form of academic misconduct in this course is **failure of the course**.

Academic Misconduct?

- **OK**

- Asking classmates general questions about the course material or concepts.
- Working with your group on Lecture or Lab problems assigned to your group.
- Writing code with a classmate for practice, as long as you are not working on assigned homework problems.
- Asking instructors for help.

- **NOT OK**

- Showing your code for a homework problem or Lab Practical to another student.
- Sharing solutions to homework or Lab Practicals with anyone not enrolled in the course.
- Taking code you didn't write (even in portions) and submitting it as your work.
- Asking anyone other than an instructor for homework help.

Policies

Late Work

- *Homework (only) will be accepted up to 24 hours late, with a 20% penalty.*

Revisions

- We allow you to revise most homeworks and Lab Practicals to get some of the points you missed back. See the Syllabus on Canvas for more details.

Excused Absences

- With documentation, you can receive some or all of the points for a lecture or lab you missed. See Canvas for more details.

First Quiz

- You MUST complete the Academic Integrity and Syllabus Quiz before you will be able to access the assignments for the course.
- The quiz is on Canvas, open course materials, not timed, and you can try as many times as you need to get 100%.
- Do it this week!

Lecture Teams

In lecture, we have group work every day – you'll sit next to your team members.

- Teams will be posted on Canvas soon.
- Bringing your laptop to class (and installing Python on it) is a good idea. Don't forget to charge it before class!

Device policy – no distractions while you're working!

If you are distracted and/or not participating,

we can award you a 0 on any problem.

Wait to check FB, snapchat, email or Pokemon Go until your group is done!

Caveat Emptor

Many students find this class challenging

- (Especially if you have no prior background)

Most important key to success: ***be proactive***

- Actively participate in class, labs, assignments
- Get help when you don't understand

Take advantage of all course resources!

- Textbook, lectures, office hours, groups ...
- 4 instructors, 65 AI/UIs – we want to help!

Announcements are Crucial!

- Are you getting the Announcements? Would you prefer them in another format?

The screenshot shows a course management system interface with a sidebar and a main content area.

Left Sidebar:

- Home
- Profile
- Notifications** (highlighted)
- Files
- Settings
- ePortfolios
- Logout
- Content Migrations

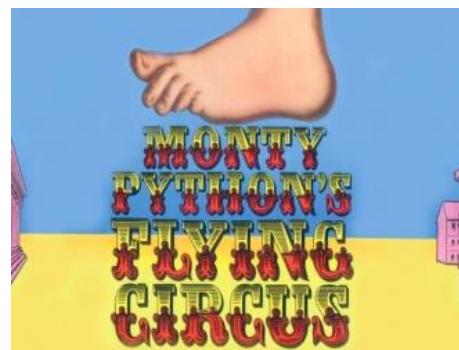
Notification Preferences Section:

Notification Preferences

Course Activities	Email Address	Cell Number	Email Address	Twitter
Due Date	kdscholl@indiana.edu	3173856307@txt.att.net	schollhouserock@gmail.com	SchollHouseRock
Grading Policies	<input type="checkbox"/> Weekly			
Course Content				
Files				
Announcement	<input checked="" type="checkbox"/> ASAP			
Announcement Created By You				
Grading	<input type="checkbox"/> Include scores when alerting about grade changes.	<input checked="" type="checkbox"/> ASAP		
Invitation	<input checked="" type="checkbox"/> ASAP			
All Submissions				
Late Grading	<input type="checkbox"/> Daily			
Submission Comment	<input type="checkbox"/> Daily			

Introducing Python

- Developed by Guido van Rossum (BDFL)
- First released in 1991
- Named after?



Why Python?

- Runs everywhere – Platform independent!
 - PC, Mac, Linux, UNIX, smart devices
 - Only requirement is that python is installed
- Easy to use, powerful!
- Free and open-source!

Python Is Easy to Use

- **High-level language**

"Programming at the speed of thought"

- Python programs are often shorter and more easily legible than those in other languages

Python Is Easy to Use (continued)

- Python Program

```
print("Game Over!")
```

- C-style Program

```
#include <iostream>

int main()
{
    std::cout << "Game Over!" << std::endl;
    return 0;
}
```

Python Is Easy to Use (continued)

- Python Program

```
import turtle

myTurtle = turtle.Turtle()
myTurtle.circle(50)
turtle.getscreen()._root.mainloop()
```

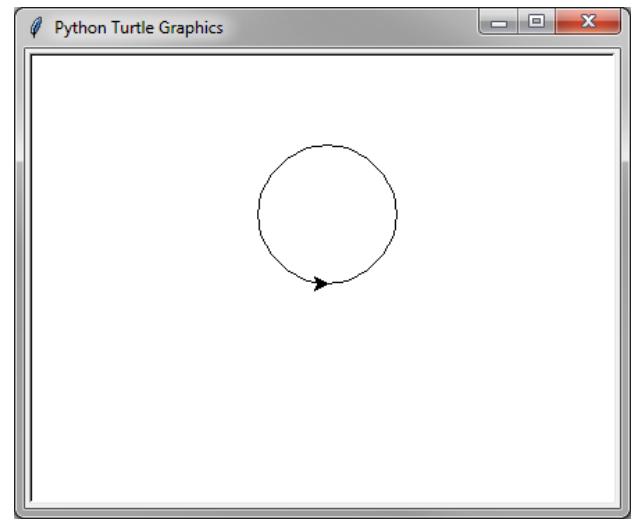
- Java Program

```
import java.awt.Graphics;
import java.applet.Applet;

public class ThreeStyles extends Applet
{
    public void paint (Graphics g)
    {
        g.drawRoundRect(40, 50, 90, 90, 200, 200);
        g.fillRoundRect(40, 160, 90, 90, 200, 200);

        g.drawOval(150, 50, 90, 90);
        g.fillOval(150, 160, 90, 90);

        g.drawArc(270, 50, 90, 90, 0, 360);
        g.fillArc(270, 160, 90, 90, 0, 360);
    }
}
```



Both of these examples draw a simple circle...

Setting up Python

To download Python:

<https://www.python.org/downloads/>

- Download the appropriate version (we suggest 3.4 or newer) for your Operating System.
- If you use Linux or MacOS, you probably already have an OLDER version of Python installed, but you want an up-to-date one!

Make sure you use the right version – STC machines may have both 2.X AND 3.X

So is Python the BEST Language?

Programming languages are *tools*.

- Is there a best tool for ANY job?
- No such thing as the “best language”



Python is easy to learn, is a very good tool for many problems, and best of all, *the concepts you learn in this class will apply to almost any other programming language!*

Computer program

Program (*noun*)

A sequence of instructions that are executed in order to accomplish a desired task.

Your job is often figuring out how to break complex task down into a series of explicit instructions.

An important step towards a program is an...

Algorithm

Algorithm (*noun*)

A finite set of precise instructions for performing a computation or solving a problem.

Example

How to add a list of numbers:

1. Start a total at 0.
2. Are there more numbers?
 - a) If so, add the next one to the total and return to Step 2.
 - b) If not, stop. The total is your answer.

*This is why we
covered algorithms
in I201!*

For Today

- We're going to do a couple of short exercises.
- Ad-hoc groups!
- We'll take a moment to group you.

Interpreted Language

Python is an *interpreted language*.

- A python program (.py file) is just a text file.

Interpreted

- Read and executed directly – *no compilation stage!*
- Examples: Python, Perl, Ruby, Lisp

Compiled

- Compiled languages are transformed into an executable form before running.
- Examples: C (and C++), Java

Introducing IDLE

Integrated DeveLopment Environment (IDLE)
= an IDE!

- IDLE ships with most Python versions

Interactive mode:

You tell Python what to do, and it will do it immediately

- *We'll start out working in this mode for CH 1 and 2*

Script mode:

You write, edit, load, and save python programs (just like you write a document in a word processor).

Jargon

Statement

- Single unit in programming language that performs some action

String: Sequence of characters

Expression: Something which has a value or that can be evaluated to a single value

Code: Sequence of programming statements

Jargon

Comment: Note in source code meant only for programmers; ignored by computer

- Start comments with #
- Use opening block of comments

Blank Lines

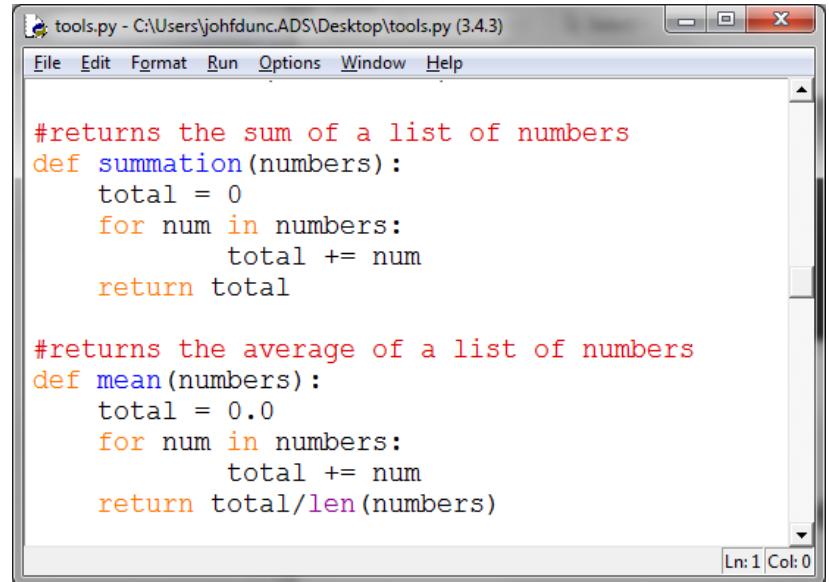
- Also (generally) ignored by computer
- Use for legibility; keep related code together

Python cares about whitespace!

- New lines (hitting Return/Enter) and tabs

Jargon

Syntax highlighting:
Displaying programming code in different colors or fonts, according to the category of each item



A screenshot of a Windows Notepad window titled "tools.py - C:\Users\johfdunc.ADS\Desktop\tools.py (3.4.3)". The window contains two functions: "summation" and "mean". The code uses color-coded syntax highlighting: red for comments, blue for keywords like "def" and "for", orange for loops and functions, and purple for the "len" function. The code is as follows:

```
#returns the sum of a list of numbers
def summation(numbers):
    total = 0
    for num in numbers:
        total += num
    return total

#returns the average of a list of numbers
def mean(numbers):
    total = 0.0
    for num in numbers:
        total += num
    return total/len(numbers)
```

The status bar at the bottom right shows "Ln: 1 Col: 0".

Bug: Error in programming code

Bugs! You will get lots of them!

Photo # NH 96566-KN First Computer "Bug", 1945

92

9/9

0800 Arctan started
1000 " stopped - arctan ✓
13" 00 (032) MP - MC { 1.2700 9.037 847 025
033) PRO 2 2.130476415 9.037 846 995 const
const 2.130676415
Relays 6-2 in 033 failed special sped test
in relay " 11.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1600 Arctangent starts.
1700 closed down.

Relay
2145
Relay 3370



Admiral Grace Hopper

Even expert programmers get bugs all the time!

Problems

Syntax errors

- *The computer doesn't understand what I wrote, because I made a typo.*
- English: “I hope you all hav a good day!”

Semantic errors

- *What I wrote doesn't make sense in that order.*
- English: “Hope day good all you have!”

You are much smarter than the computer!

To do for next time

1. Set Python (3.4 or newer) up on your home computer or laptop.
2. Read CH1 and start CH2 of the textbook.
3. Fill out the GROUP FORMATION SURVEY:
 - <https://goo.gl/forms/E2dk6NbKiXwUi7I22>
 - We need your answers by Tuesday!

Questions?

INFORMATICS

Lecture 2

I210 – INTRODUCTION TO
PROGRAMMING WITH PYTHON

Today

- **Mathematical operators**
- **Functions**
- **Variables, integers & booleans**

If you miss anything today, please note that all slides from lecture are posted to Canvas on Tuesday and Thursday afternoons.

Lecture Teams

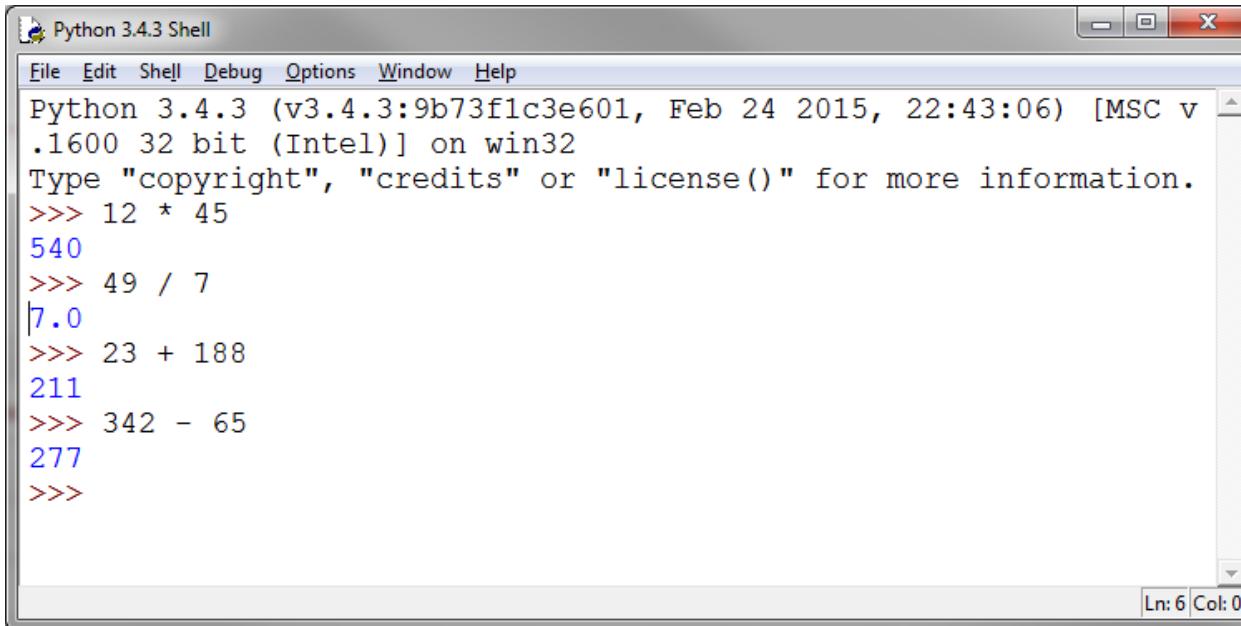
- Posted to Canvas
- If your lecture is using assigned seating for groups, this information is also posted.
- Make sure your group gets checked off when you're done with a problem! (At that point, you're allowed to use your devices to check email, etc).

IDLE: Interactive Mode

IDLE's initial window

Make sure you have 3.X

You can type in an expression and get an immediate result (**great for testing!**):



The screenshot shows the Python 3.4.3 Shell window. The title bar reads "Python 3.4.3 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main window displays the Python interpreter's prompt and several calculations:

```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 12 * 45
540
>>> 49 / 7
7.0
>>> 23 + 188
211
>>> 342 - 65
277
>>>
```

In the bottom right corner of the window, there is a status bar with "Ln: 6 Col: 0".

Working with Numbers

Numeric types in Python

Integers: Numbers without a decimal part

1, 0, 27, -100

Floats (Floating-Point Numbers): Numbers with a decimal part (even if it's .0)

2.376, -99.1, 1.0

Mathematical Operators

Operator	Description	Example	Evaluates To
<code>+</code>	Addition	<code>7 + 3</code>	10
<code>-</code>	Subtraction	<code>7 - 3</code>	4
<code>*</code>	Multiplication	<code>7 * 3</code>	21
<code>**</code>	Exponentiation	<code>7 ** 3</code>	343
<code>/</code>	Division	<code>7 / 3</code>	2.33333333...
<code>//</code>	Integer by Integer Division (rounds down)	<code>7 // 3</code>	2
<code>%</code>	Modulus (remainder under integer division)	<code>7 % 3</code>	1

Modulus

Modulus gives the remainder under division

10 mod 3 → 1

means "*what is the integer remainder when 10 is divided by 3?*"

10 mod 2 → 0

because 2 evenly divides 10.

In Python, we use % for mod. **10 % 2 → 0**

Functions

Function: A named collection of programming code that can receive values, do some work, and return values

- A **function** is like a delivery restaurant:
 - Make a call : *I'd like to order a curry*
 - Provide information: *I like chicken, spiciness 2*
 - Get something back: *your food*

Functions

Values passed *into* a function are called **arguments**.

Values sent *back* by a function are called **return values**.

```
>>> abs(-10)
```

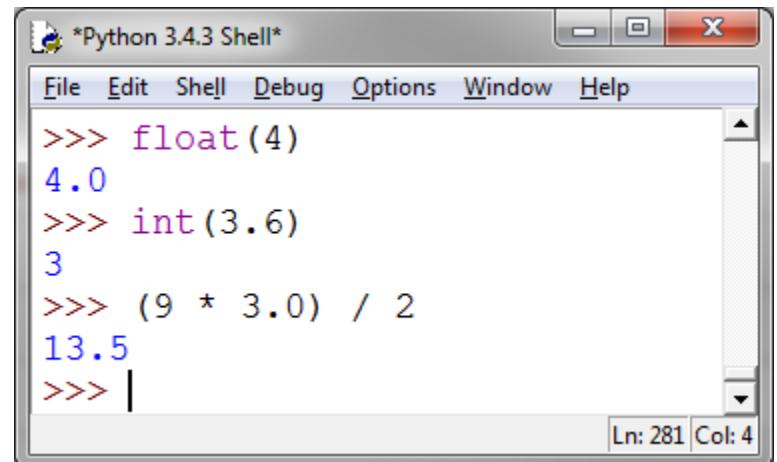
```
10
```

- **abs()** is a function that takes **a number** as an argument, and returns **the absolute value** of that **number**.

float() and int() functions

To convert an integer into a float, we can use the **float()** function OR multiply/divide by a decimal number.

- If we use the **int()** function on a float, it will chop off the decimal part!
- A calculation that involves a float will also return a float.



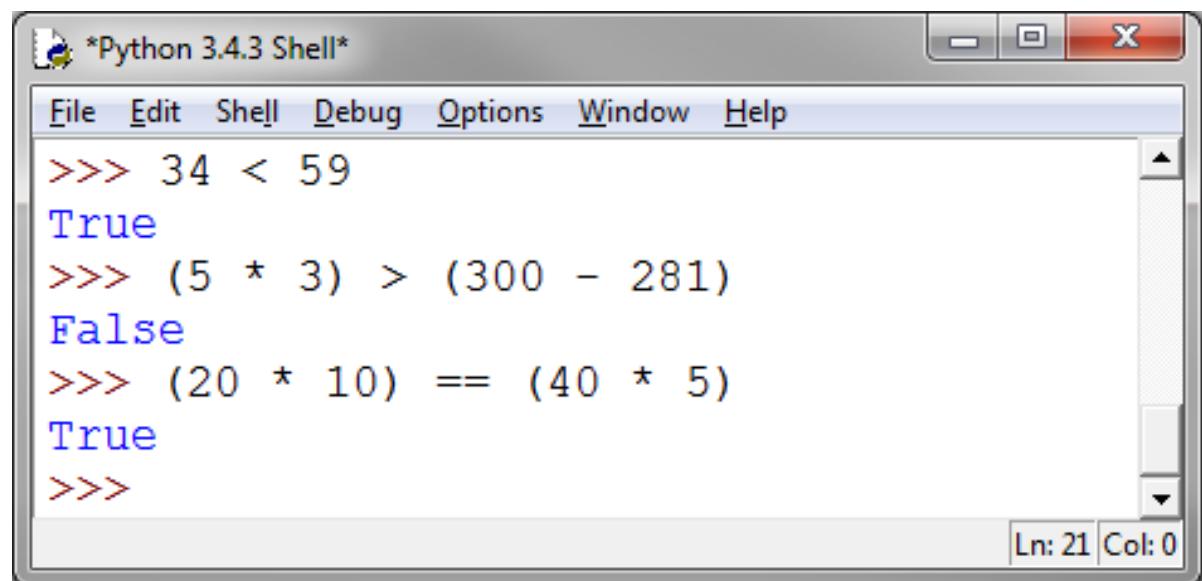
```
*Python 3.4.3 Shell*
File Edit Shell Debug Options Window Help
>>> float(4)
4.0
>>> int(3.6)
3
>>> (9 * 3.0) / 2
13.5
>>> |
Ln: 281 Col: 4
```

Boolean Values

Boolean values

In addition to a numeric value, an expression can resolve to either **True** or **False**.

- We have a number of operators that express these relationships:



The screenshot shows a window titled '*Python 3.4.3 Shell*' with the following content:

```
File Edit Shell Debug Options Window Help
>>> 34 < 59
True
>>> (5 * 3) > (300 - 281)
False
>>> (20 * 10) == (40 * 5)
True
>>>
```

The window has standard operating system window controls (minimize, maximize, close) at the top right. A status bar at the bottom right indicates 'Ln: 21 Col: 0'.

Boolean Comparison Operators

Operator	Name	Example	Truth Value
<code>==</code>	Equals	<code>5 == 6</code>	False
<code>!=</code>	Not Equal	<code>5 != 6</code>	True
<code>></code>	Greater Than	<code>5 > 6</code>	False
<code>>=</code>	Greater Than or Equal	<code>5 >= 6</code>	False
<code><</code>	Less Than	<code>5 < 6</code>	True
<code><=</code>	Less Than or Equal	<code>5 <= 6</code>	True

* Be sure to note that we use `==` for equality. `=` has a different meaning.

Variables

Variables store information in your program

- A variable is a placeholder for values
- You can assign different values to variables, and you change the value of a variable over time

Example

price 26

Variables

Variables store information in your program

- A variable is a placeholder for values
- You can assign different values to variables, and you change the value of a variable over time

Example

price 40

Variables

Variable names are **unique** and **case-sensitive**

- **name** and **nAme** are NOT the same!!

A variable contains **data that has a type**

(e.g. integer, string)

- Python does not require you to declare that type when declaring a variable – it figures it out.
- You can even change the value of a variable to a different type of data!

Variable Definitions

A variable is created using a **declaration**:

my_number = 3

The value **3** here is the **initial value** of the variable.

A variable is **assigned** further values using **=**
(assignment operator)

my_number = 6 * 5

Naming Variables

Legal variable names:

- Contain only numbers, letters, and underscores
- But don't start with a number
 - Starting with _ means something special (for now, don't)
- Can't use certain words (**reserved or keywords**)

Keywords in Python 3.4:

- | | | | | |
|----------|-------|--------|----------|---------|
| – False | None | True | and | as |
| – assert | break | class | continue | def |
| – del | elif | else | except | finally |
| – for | from | global | if | import |
| – in | is | lambda | nonlocal | not |
| – or | pass | raise | return | try |
| – while | with | yield | | |

Naming Variables

Good naming:

- Is descriptive! (**name** vs **n**)
 - Is consistent! (**newPrice** or **new_price**, pick one)
 - Observes conventions (start with lowercase letter)
 - Isn't too long (try for no more than 15 characters)
-
- Remember, case matters!!!
 - **my_name** and **my_nAme** ARE NOT the same!!

Example - Calculating GPA

GPA is your total number of credits divided by your total number of credit hours.

We can calculate this with a few variables and the division operator:

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> total_credits = 185
>>> total_hours = 53
>>> total_credits / total_hours
3.490566037735849
>>> |
Ln: 115 Col: 4
```

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> total_credits = 198
>>> total_hours = 51
>>> total_credits / total_hours
3.8823529411764706
>>> |
Ln: 126 Col: 4
```

Goal

One of our most important goals:

Self-Documenting Code!

- This means your code is so legible, you can use only *minimal* comments.
- Choosing good variable names is a key step!

time_in_min = num_hours * 60

(The names make this step clear.)

Compound Boolean Conditions

We'll often want to write more complicated Boolean Conditions.

E.g. For a computer security system:

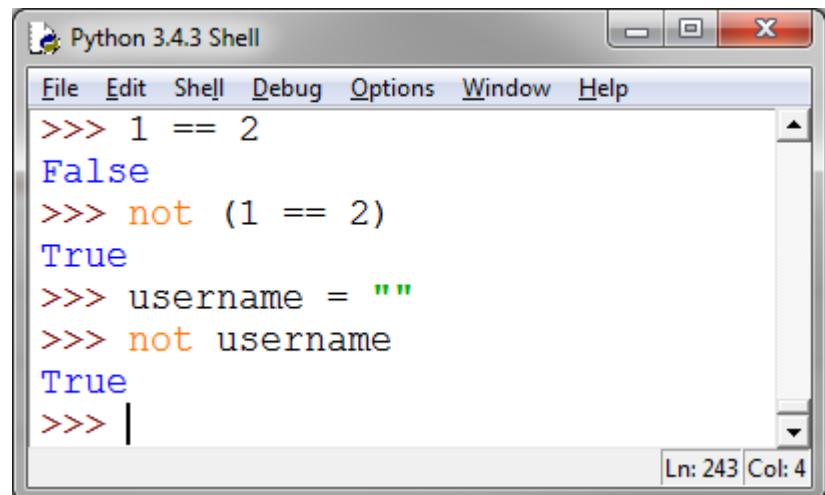
- Display an error message if a user does **NOT** enter their username
- Allow user to log in if username **AND** password are correct
- Notify staff if a break-in is detected **OR** if an incorrect password is entered 3 times

NOT

not is a logical operator that reverses the value of any Boolean Condition

TAKE NOTE:

The values **0**, **""**, and
None are also considered
to be **False**.



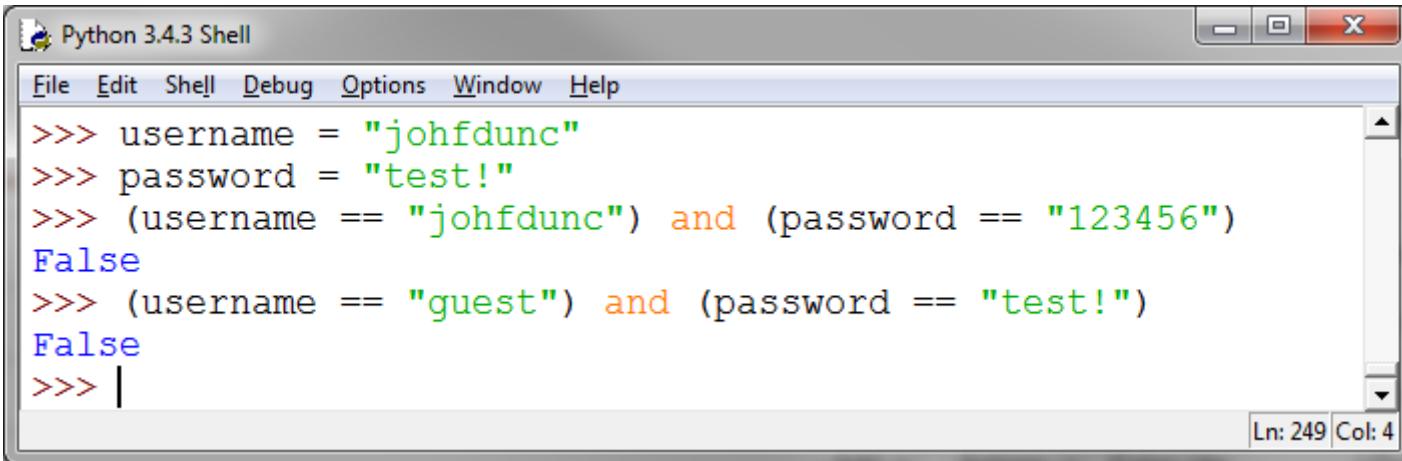
A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area shows the following interactions:

```
>>> 1 == 2
False
>>> not (1 == 2)
True
>>> username = ""
>>> not username
True
>>> |
```

The status bar at the bottom right indicates Ln: 243 Col: 4.

AND

and is a logical operator that connects two Boolean Conditions. It's **True** if they are BOTH true, and **False** otherwise.



The screenshot shows a Windows-style window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays Python code and its output:

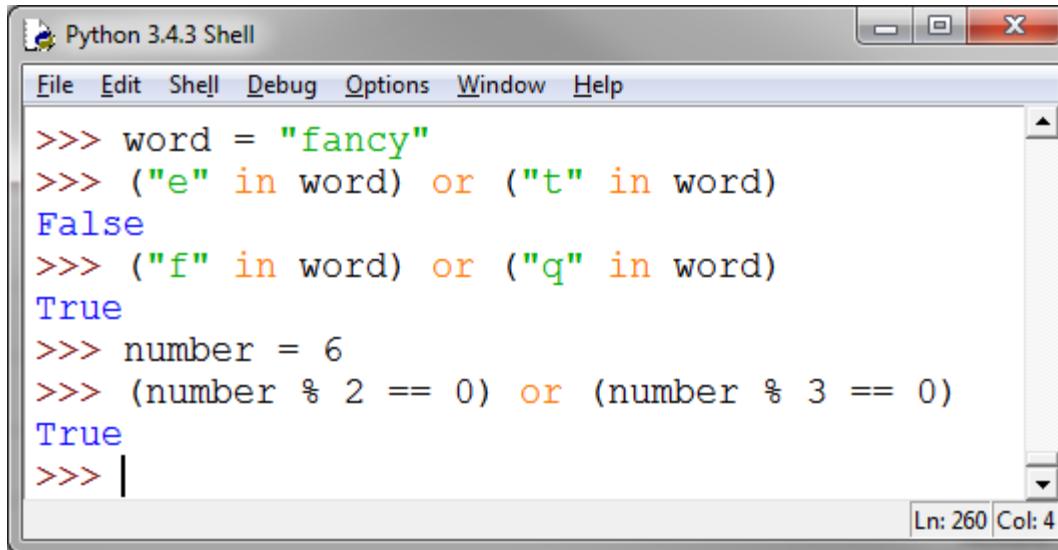
```
>>> username = "johfdunc"
>>> password = "test!"
>>> (username == "johfdunc") and (password == "123456")
False
>>> (username == "guest") and (password == "test!")
False
>>> |
```

In the bottom right corner of the window, there is a status bar with "Ln: 249 Col: 4".

- If I have the wrong username or the wrong password...

OR

or is a logical operator that connects two Boolean Conditions. It's true if either of them is true OR if they both are!



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area contains the following Python code:

```
>>> word = "fancy"
>>> ("e" in word) or ("t" in word)
False
>>> ("f" in word) or ("q" in word)
True
>>> number = 6
>>> (number % 2 == 0) or (number % 3 == 0)
True
>>> |
```

The status bar at the bottom right indicates "Ln: 260 Col: 4".

Question: Is this the way we use OR in English?

XOR

xor, or exclusive or, is a common CS operator which behaves like the ‘or’ we know in English:

- “It will rain tomorrow or it will not” – Can both be true?
- True xor True = False
- False xor False = False
- True xor False = True

From
I201:

p	q	$p \vee q$	$p \oplus q$
F	F	F	F
F	T	T	T
T	F	T	T
T	T	T	F

Python doesn’t implement this as **xor**, but you can do it like this:

boolean_condition1 != boolean_condition2

Precedence

variableA OR variableB AND variableC

If this ever comes up, this is the order of precedence, from most to least:

1. Not
2. And
3. Or

Lesson – Use parenthesis!!

time_out or (correct_id and correct_pw)

Questions?

INFORMATICS

Lecture 3

I210 – INTRODUCTION TO
PROGRAMMING WITH PYTHON

Today

Announcement: PLTL Office Hours

- **strings** (and concatenation) – *New data type*
- **in operator**
- **len function**
- **indexing strings**
- **lists** (changing, deleting, min/max/sum, appending, removing, sorting)
- **methods versus functions**

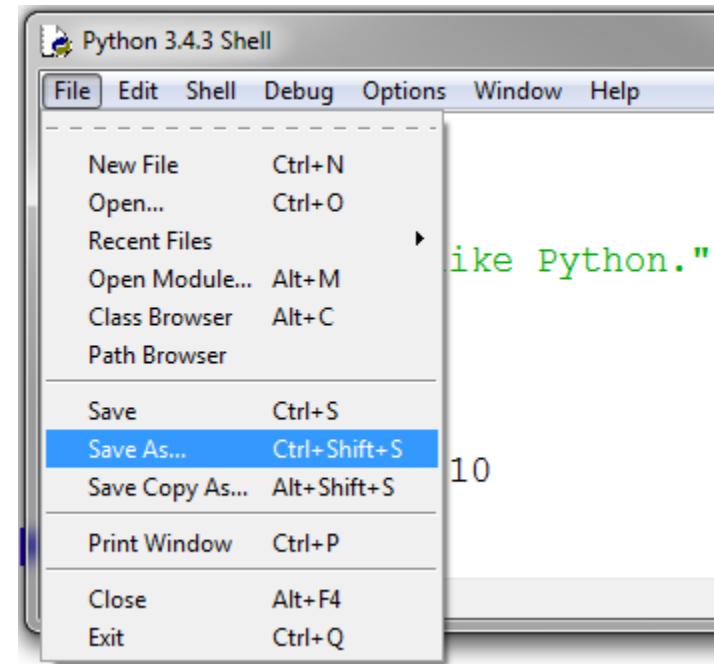
PLTL Office Hours

PLTL Leader	Office Hour Location	Office Hour Time and Date
Coming Soon		
Nicholas Graham nicgraha@indiana.edu		

Saving to a File

In IDLE's Interactive Mode,
you can save a copy
of your session:

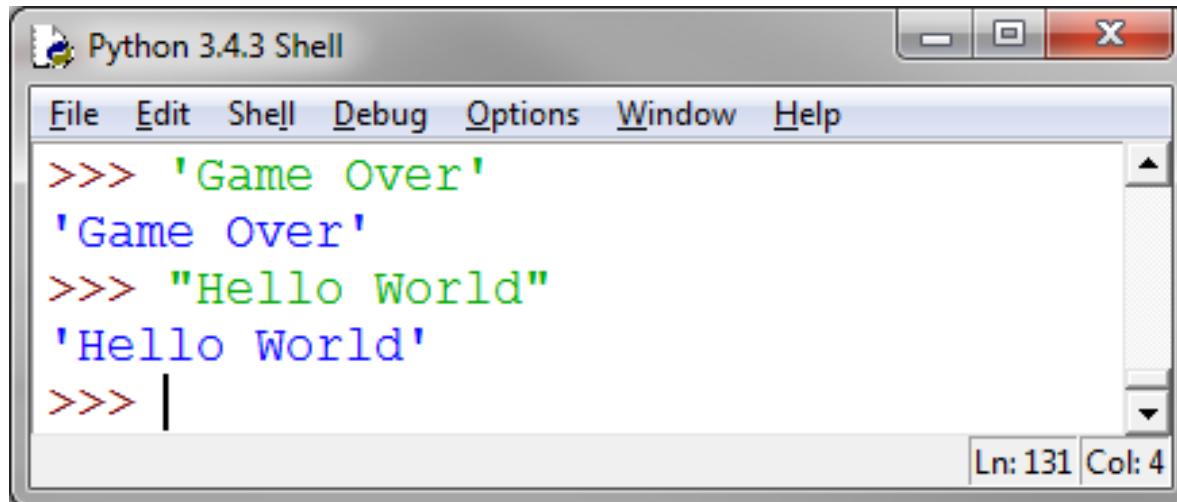
- Save it as a text file (.txt file extension)
- This saves a record of the entire session. It won't allow you to re-run the code, however.



Strings: Using Quotes

Strings (text values) in Python use quotes.

- Either single quotes ' or double quotes " work!
- Start *and* end with the same quotes:



A screenshot of the Python 3.4.3 Shell window. The title bar reads "Python 3.4.3 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main window displays the following Python session:

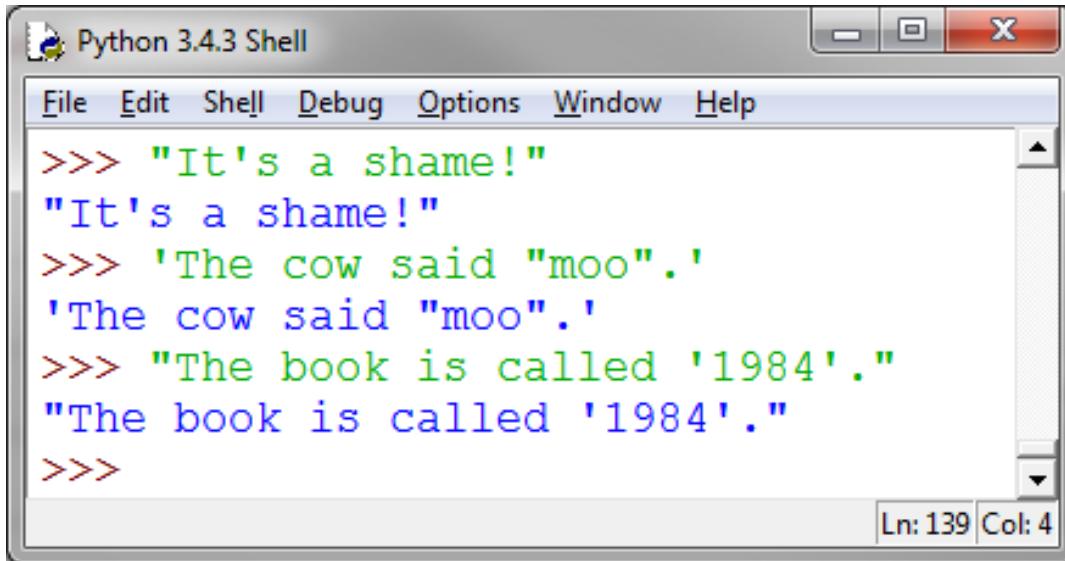
```
>>> 'Game Over'
'Game Over'
>>> "Hello World"
'Hello World'
>>> |
```

The status bar at the bottom right shows "Ln: 131 Col: 4".

Strings: Using Quotes

Why choose one quote over the other?

- If your string includes one type of quote (or apostrophe), use the other one:



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python code and its output:

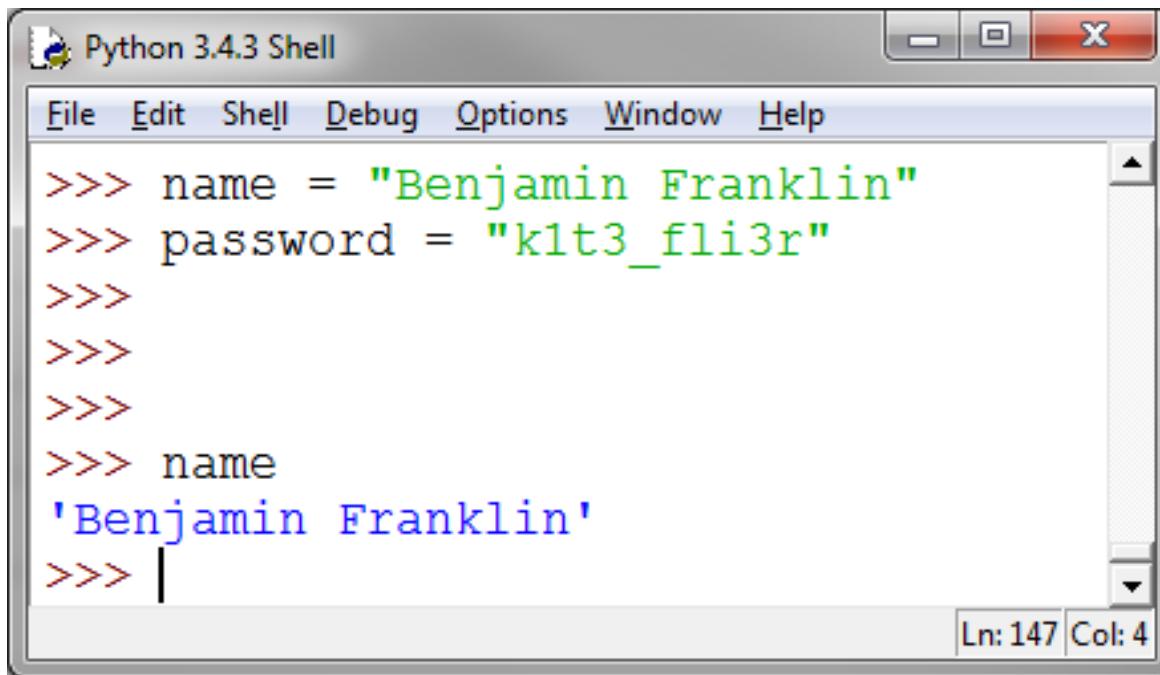
```
>>> "It's a shame!"  
"It's a shame!"  
>>> 'The cow said "moo".'  
'The cow said "moo".'  
>>> "The book is called '1984'."  
"The book is called '1984'."  
>>>
```

The status bar at the bottom right indicates "Ln: 139 Col: 4".

*IDLE defaults to single quotes, but I prefer **double quotes***

Strings

Strings can be stored in variables also:



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area displays the following Python session:

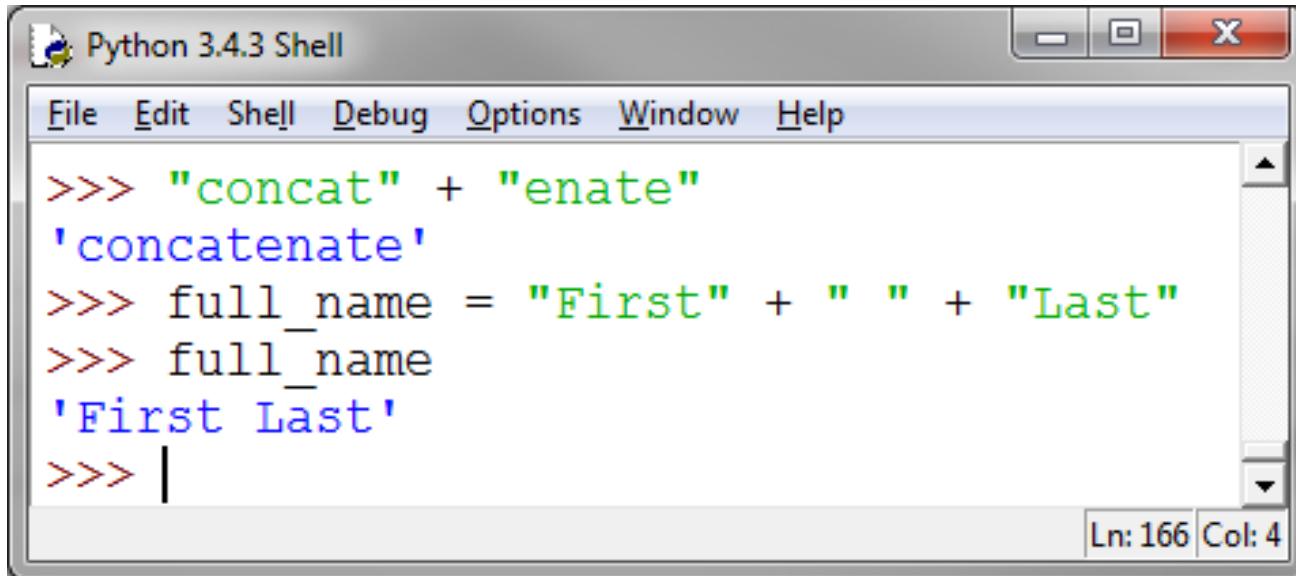
```
>>> name = "Benjamin Franklin"
>>> password = "k1t3_fli3r"
>>>
>>>
>>>
>>> name
'Benjamin Franklin'
>>> |
```

In the bottom right corner of the shell window, there is a status bar with "Ln: 147 Col: 4".

String Concatenating

Concatenation: Joining together of two pieces to form a larger whole

- In Python, the string concatenation operator is also **+**



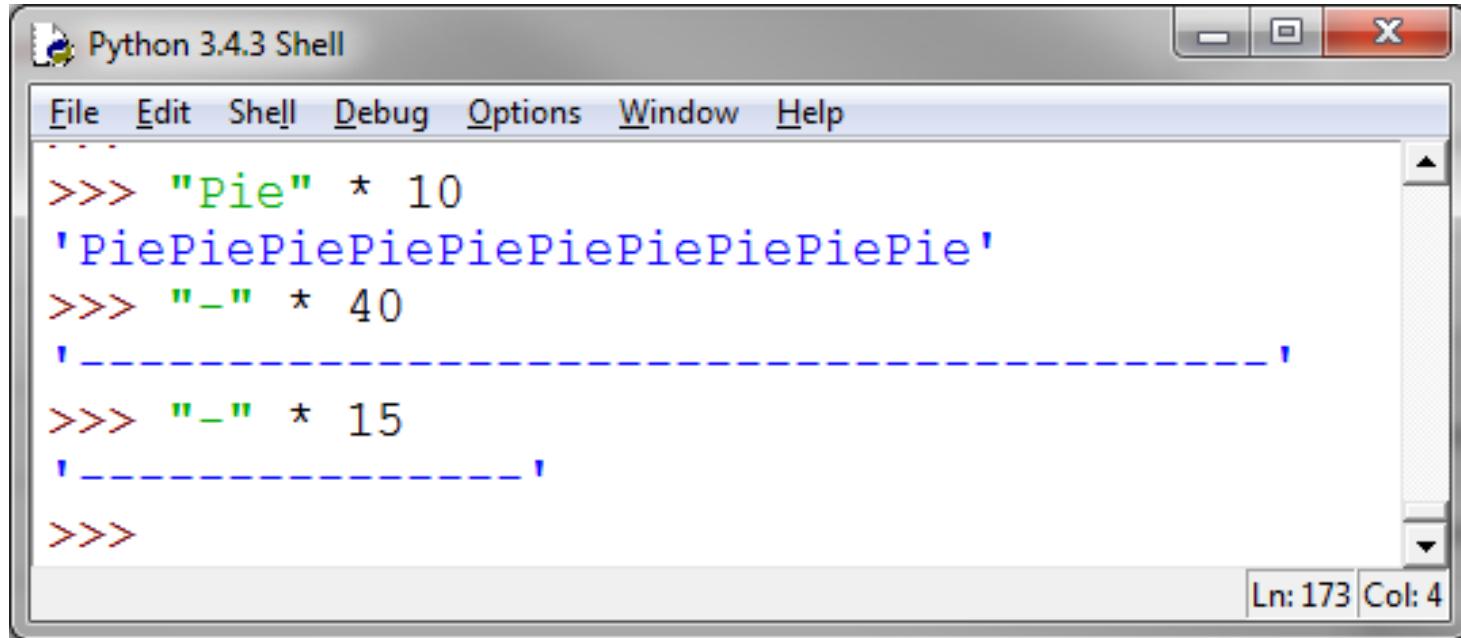
The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays Python code and its output:

```
>>> "concat" + "enate"
'concatenate'
>>> full_name = "First" + " " + "Last"
>>> full_name
'First Last'
>>> |
```

In the bottom right corner of the shell window, there is a status bar with the text "Ln: 166 Col: 4".

Repeating Strings

The ***** operator creates a new string by concatenating a string a specified number of times:

A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window shows the following code and output:

```
...>>> "Pie" * 10
'PiePiePiePiePiePiePiePiePiePie'
>>> "_" * 40
'-----'
>>> "_" * 15
'-----'
>>>
```

A status bar at the bottom right indicates "Ln: 173 Col: 4".

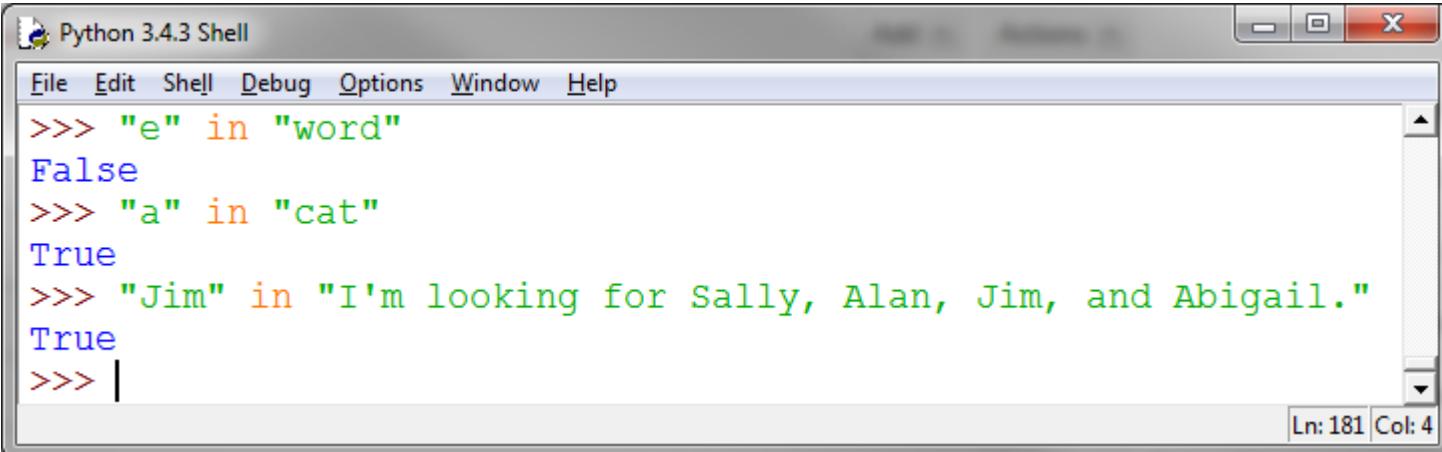
The screenshot displays three examples of the multiplication assignment operator (*). The first example, "Pie" * 10, results in the string 'PiePiePiePiePiePiePiePiePiePie'. The second example, "_" * 40, results in the string '-----'. The third example, "_" * 15, results in the string '-----'. The status bar at the bottom right of the shell window shows "Ln: 173 Col: 4".

This will be useful later!

in operator

The **in** operator allows us to see if something is found in a collection of things.

- We can test for a string being a **substring** of another string:



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following code and output:

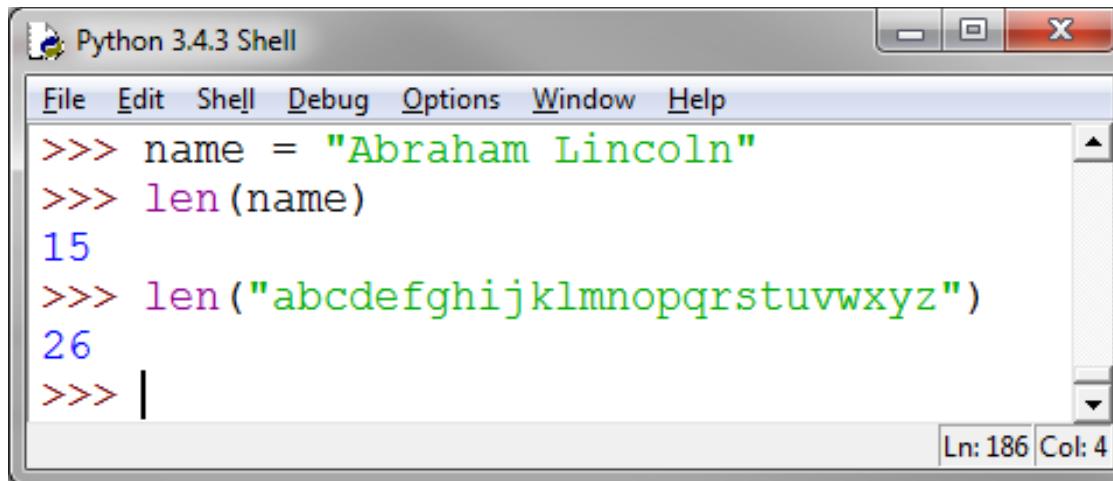
```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> "e" in "word"
False
>>> "a" in "cat"
True
>>> "Jim" in "I'm looking for Sally, Alan, Jim, and Abigail."
True
>>> |
```

The status bar at the bottom right shows Ln: 181 Col: 4.

len function

The **len** function (short for **length**) counts the number of pieces in a collection.

- A string is a collection of characters, so it tells us how many characters (including spaces) are in a string:



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area shows the following interactions:

```
>>> name = "Abraham Lincoln"
>>> len(name)
15
>>> len("abcdefghijklmnopqrstuvwxyz")
26
>>> |
```

The status bar at the bottom right indicates Ln: 186 Col: 4.

Indexing strings

word = "learn"

L	E	A	R	N
0	1	2	3	4

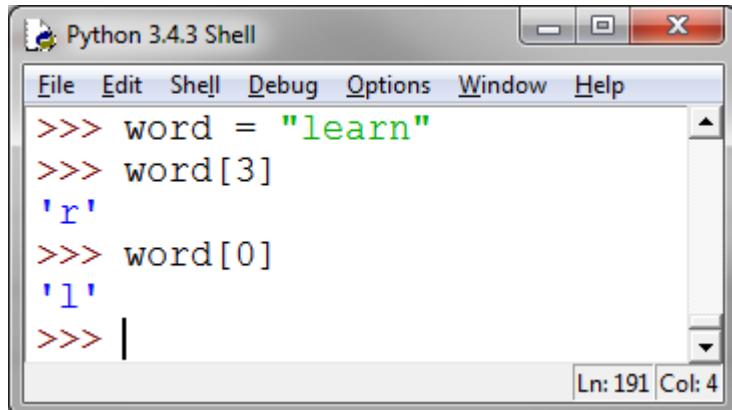
- Each letter in a string has a numeric position called an **index**.
- This numbering starts at **0**, not **1**!!

Indexing strings

word = "learn"

L	E	A	R	N
0	1	2	3	4

- We can refer to these by using **indexing**:



The screenshot shows a Windows-style window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area contains the following Python session:

```
>>> word = "learn"
>>> word[3]
'r'
>>> word[0]
'l'
>>> |
```

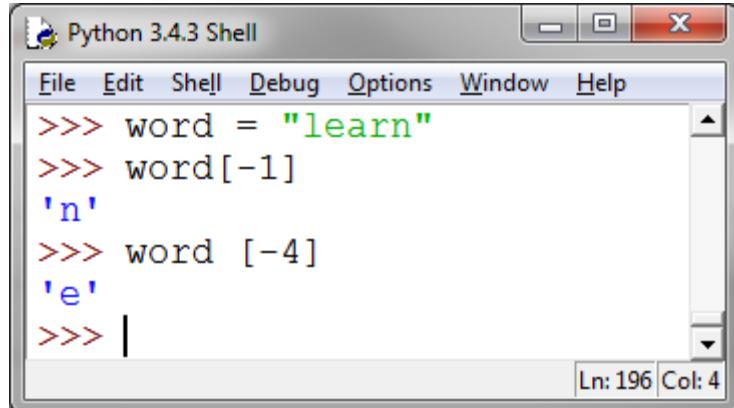
In the bottom right corner of the window, it says "Ln: 191 Col: 4".

This is also called
indexing into
something.

Negative Indexing

L	E	A	R	N
-5	-4	-3	-2	-1

- Sometimes, we care more about the *end of a string*.
- We can use **negative indexing** to quickly refer to the characters in a string, starting at the end:



The screenshot shows a Python 3.4.3 Shell window. The code entered is:

```
>>> word = "learn"
>>> word[-1]
'n'
>>> word [-4]
'e'
>>> |
```

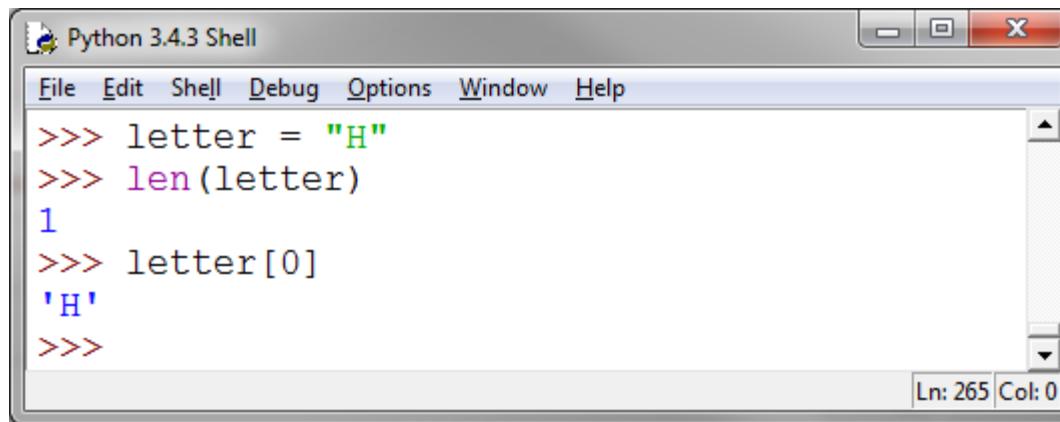
The output shows the character 'n' at index -1 and the character 'e' at index -4. The status bar at the bottom right indicates "Ln: 196 Col: 4".

len()

```
>>> len( "I like Python." )
```

14

Remember, **len** counts 1 character as length 1, but the **index** to that position is 0.



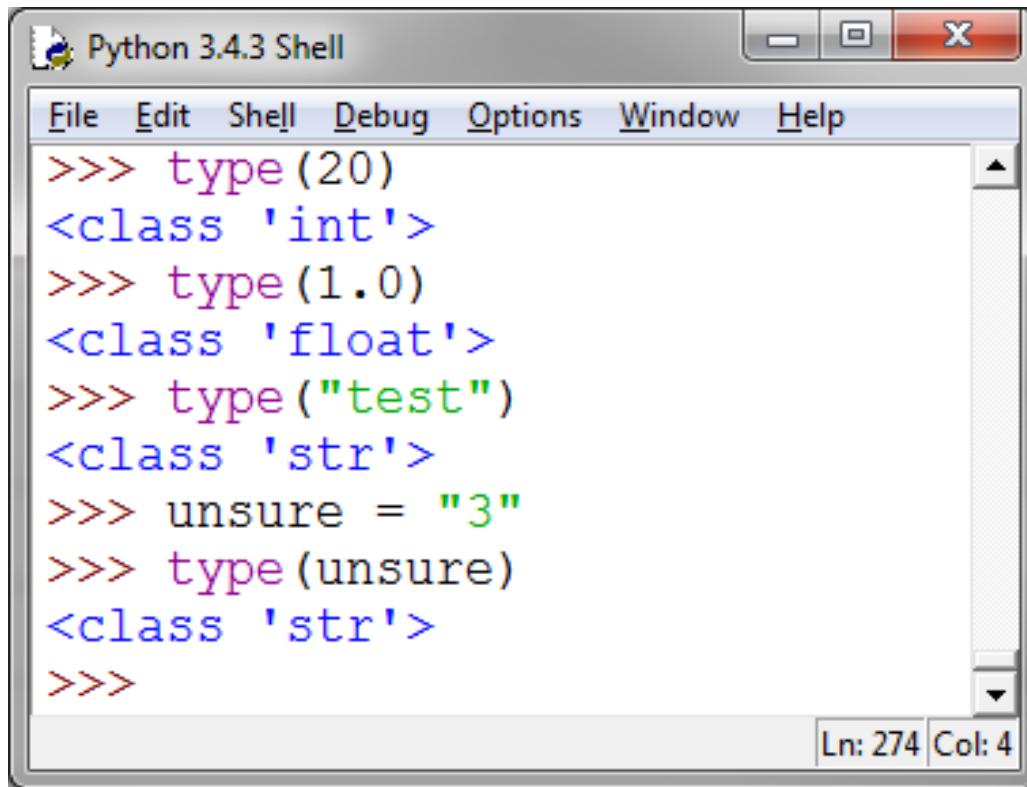
A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following code and its execution:

```
>>> letter = "H"
>>> len(letter)
1
>>> letter[0]
'H'
>>>
```

The status bar at the bottom right shows "Ln: 265 Col: 0".

The type function

If you're ever unsure what kind of data you have, you can use the **type** function:



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following code and its output:

```
>>> type(20)
<class 'int'>
>>> type(1.0)
<class 'float'>
>>> type("test")
<class 'str'>
>>> unsure = "3"
>>> type(unsure)
<class 'str'>
>>>
```

The status bar at the bottom right shows Ln: 274 Col: 4.

2 More Types of Errors

Logical Errors

- All the syntax is correct
- All the semantics are correct
- Program does not perform correctly

Runtime Error

- The program begins to execute, then crashes
- These will happen more when we learn to give direct input to our programs

Understanding Sequences

Sequence: An *ordered* list of elements

Element: A single item in a sequence

Iterate: To move through a sequence, *in order*

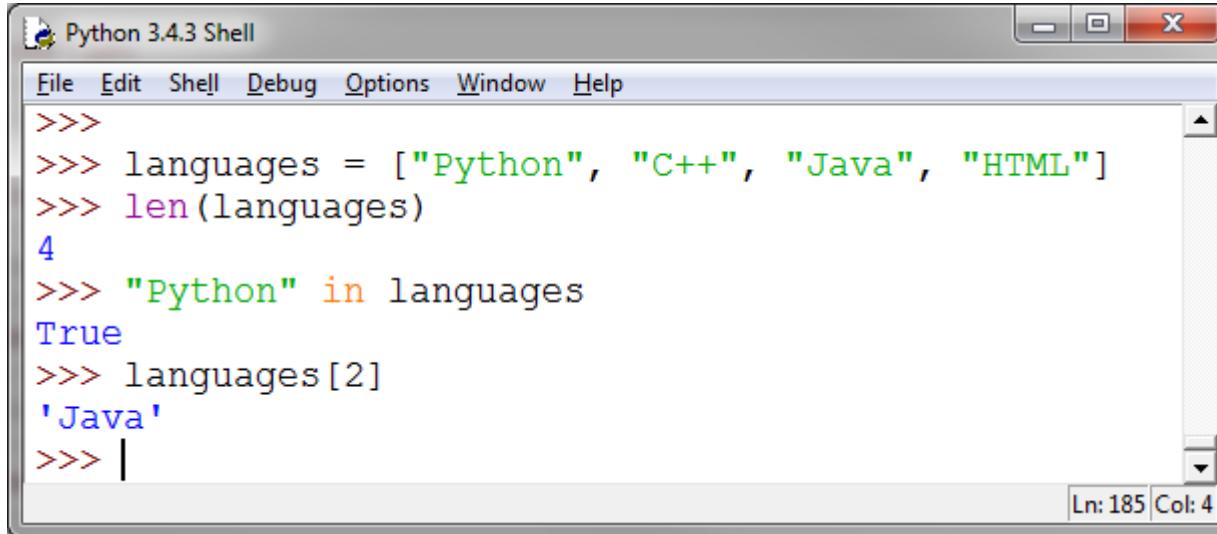
Example: The 10 best movies this year

- This is a sequence because it has an **order**
- The elements are the **movies**
- We could iterate over it **1 to 10**, or **10 to 1**

Lists - Creating

Lists in Python are a data type for storing sequences. Lists use square brackets: []

- In other languages, they might be called *vectors*.
- Notice how **len()**, the **in** operator, and indexing work:



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area displays the following Python session:

```
>>>
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> len(languages)
4
>>> "Python" in languages
True
>>> languages[2]
'Java'
>>> |
```

In the bottom right corner of the window, there is a status bar with "Ln: 185 Col: 4".

Augmented Assignment Operators

Often, we assign a value to a variable based on its original value, or concatenate a new piece on.

Augmented assignment operators:

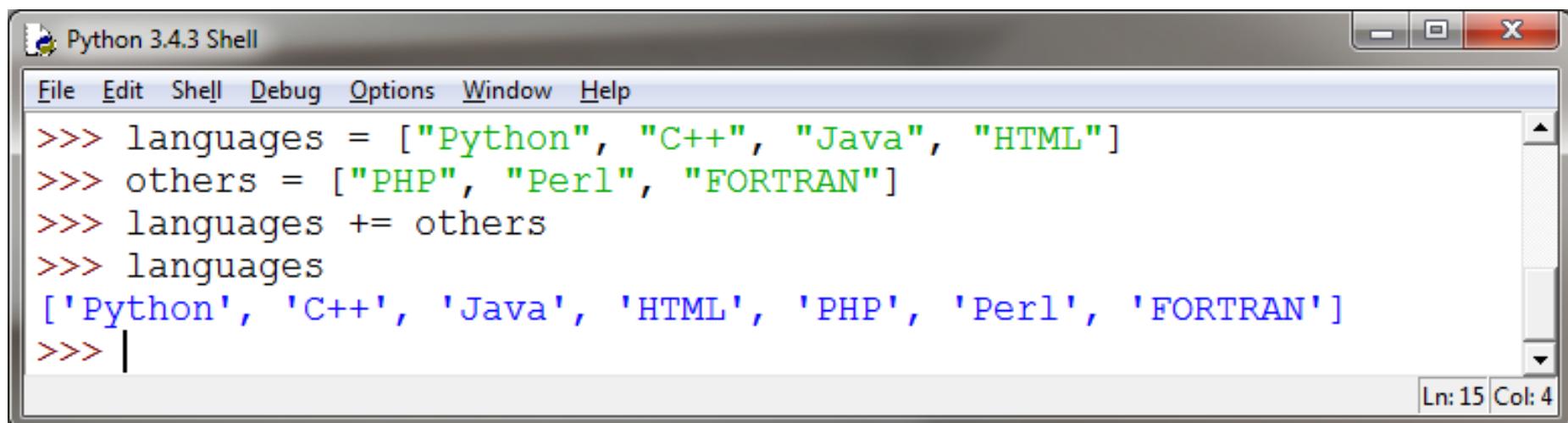
Original: **score = score + 1**

Augmented: **score += 1**

Operator	Example	Is Equivalent To
<code>*=</code>	<code>x *= 5</code>	<code>x = x * 5</code>
<code>/=</code>	<code>x /= 5</code>	<code>x = x / 5</code>
<code>%=</code>	<code>x %= 5</code>	<code>x = x % 5</code>
<code>+=</code>	<code>x += 5</code>	<code>x = x + 5</code>
<code>-=</code>	<code>x -= 5</code>	<code>x = x - 5</code>

Lists – Concatenating (adding)

Lists **add** like strings and tuples:



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area contains the following Python code:

```
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> others = ["PHP", "Perl", "FORTRAN"]
>>> languages += others
>>> languages
['Python', 'C++', 'Java', 'HTML', 'PHP', 'Perl', 'FORTRAN']
>>> |
```

The output shows the resulting list after concatenation. The status bar at the bottom right indicates "Ln: 15 Col: 4".

Lists - Changing Elements

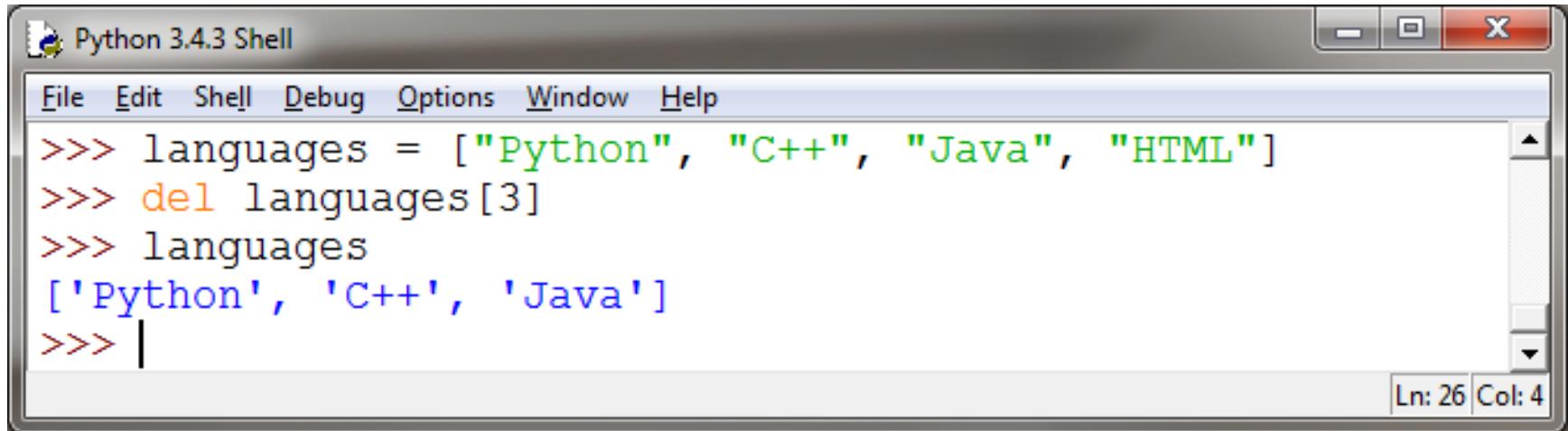
Make sure you don't try to change elements that don't exist!!



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> languages[4] = "Perl"
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    languages[4] = "Perl"
IndexError: list assignment index out of range
>>>
Ln: 22 Col: 4
```

Lists – Deleting Elements

We can erase a portion of the list by using the keyword **del**:



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area shows the following Python session:

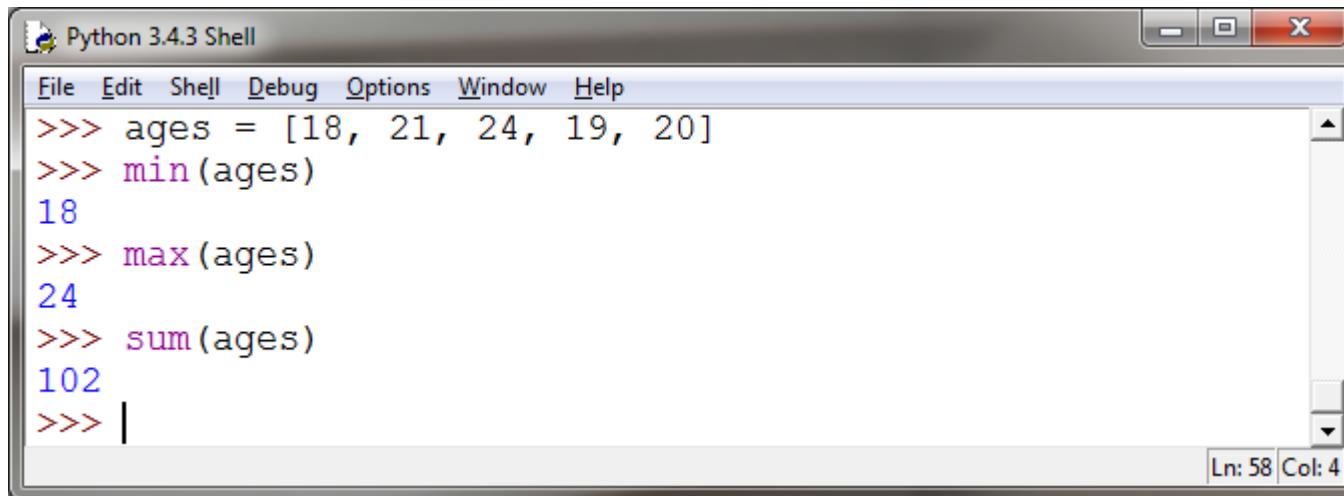
```
File Edit Shell Debug Options Window Help
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> del languages[3]
>>> languages
['Python', 'C++', 'Java']
>>> |
```

The status bar at the bottom right indicates "Ln: 26 Col: 4".

del

Lists – min, max, sum

Python has three very useful functions for working with numeric lists:



The screenshot shows a Windows-style window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area contains the following Python code:

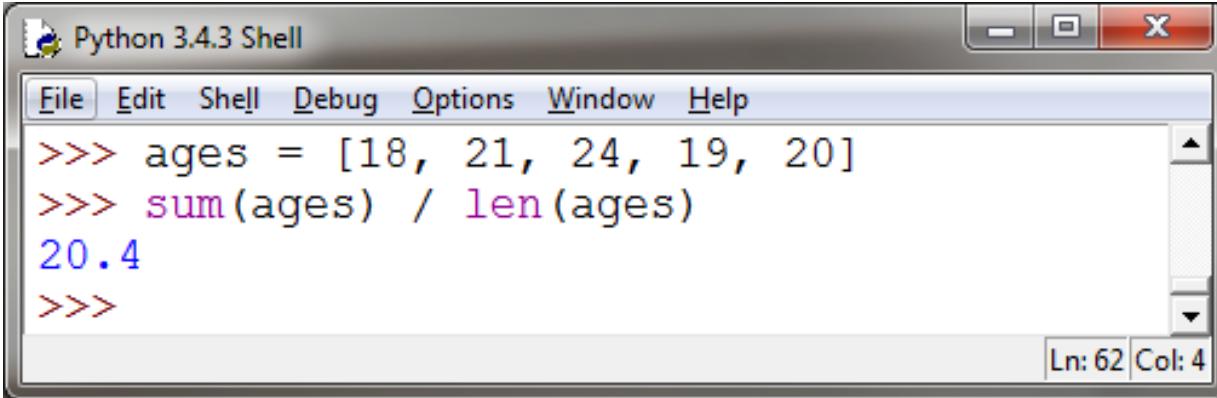
```
>>> ages = [18, 21, 24, 19, 20]
>>> min(ages)
18
>>> max(ages)
24
>>> sum(ages)
102
>>> |
```

In the bottom right corner of the window, there is a status bar with "Ln: 58 Col: 4".

min, max, sum

Lists – min, max, sum

Notice how short a program we can write to solve the problem of calculating the average age of a group:



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main window displays the following Python code and its output:

```
>>> ages = [18, 21, 24, 19, 20]
>>> sum(ages) / len(ages)
20.4
>>>
```

In the bottom right corner of the shell window, there is a status bar with "Ln: 62 Col: 4".

- Good code makes use of existing functionality where appropriate.

Functions and Methods

The words **method** and **function** are often used interchangeably by programmers.

- Let's use ***method*** to refer to a specific type of function in Python – one that's tied to an **object**.
 - *Objects are things like strings, or files, or lists*

Methods vs Functions

We call a **method** like this:

`object_name.method_name()`

 Notice the dot!

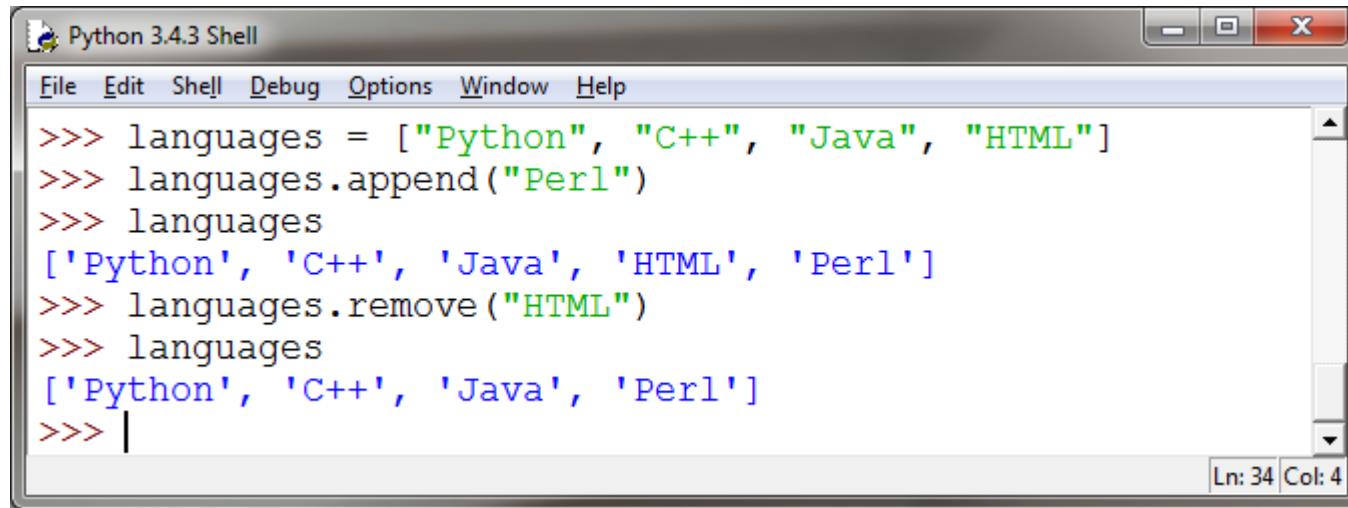
Functions, like `float()` are called on their own:

`>>> float(3)`

 No dot!

Lists – Appending and Removing

The **.append()** method is the preferred way to add things to a list! It will add the new element to the end.



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area shows the following Python session:

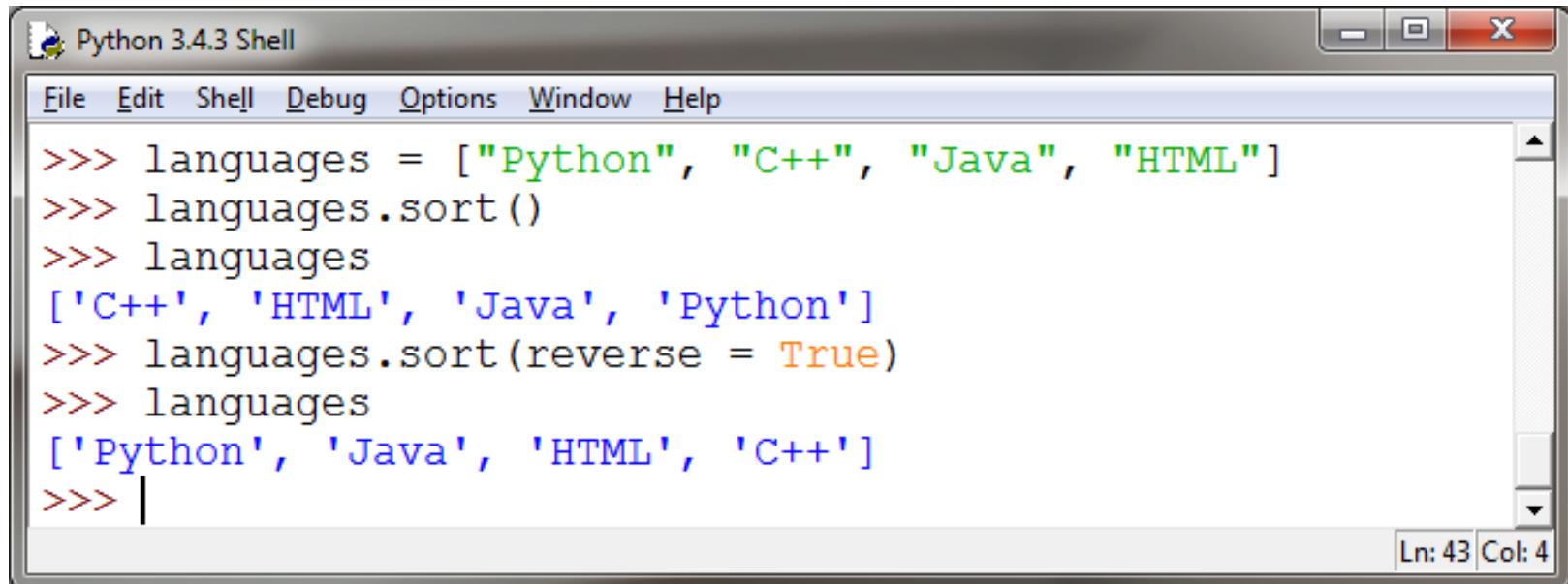
```
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> languages.append("Perl")
>>> languages
['Python', 'C++', 'Java', 'HTML', 'Perl']
>>> languages.remove("HTML")
>>> languages
['Python', 'C++', 'Java', 'Perl']
>>> |
```

The status bar at the bottom right indicates Ln: 34 Col: 4.

The **.remove()** method removes an element from the list based on its value.

Lists - Sorting

We can **.sort()** a list in ascending or descending order:



The screenshot shows the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area contains the following Python code:

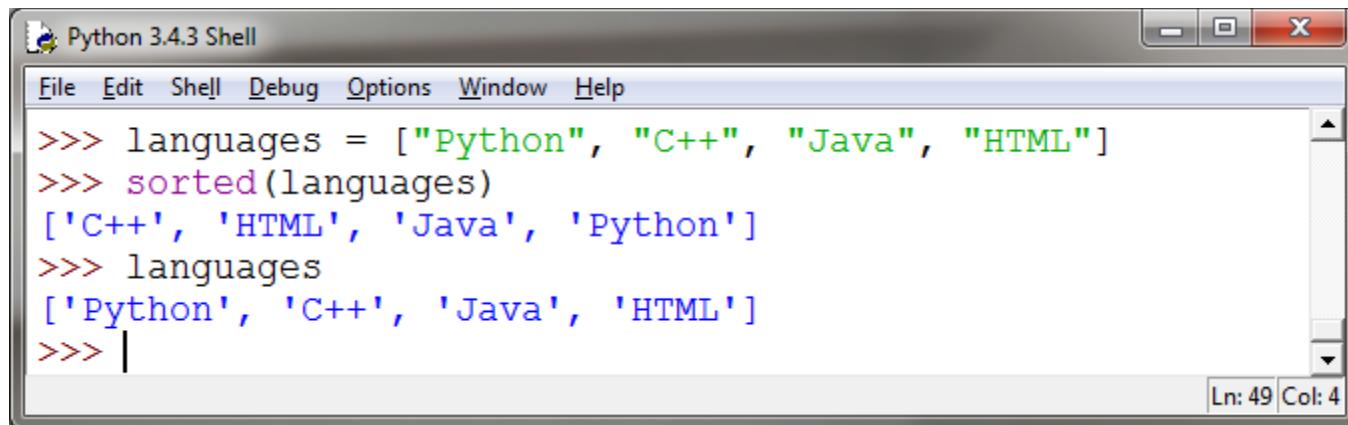
```
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> languages.sort()
>>> languages
['C++', 'HTML', 'Java', 'Python']
>>> languages.sort(reverse = True)
>>> languages
['Python', 'Java', 'HTML', 'C++']
>>> |
```

The status bar at the bottom right indicates Ln: 43 Col: 4.

Be careful! These methods *change the original list!*

Lists - Sorting

If you don't want to change the original list, use **sorted(list_name)**, which returns a sorted copy, but doesn't change the original.



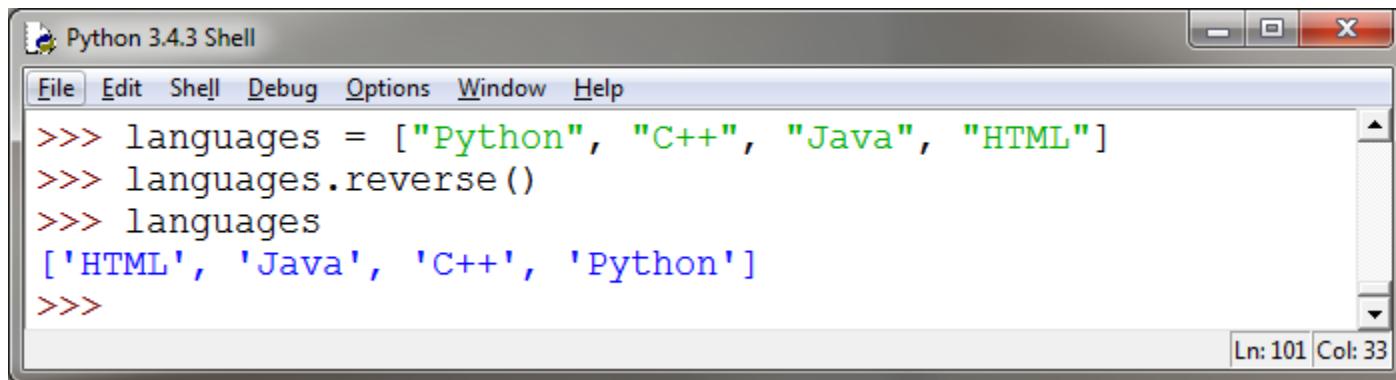
The screenshot shows a Windows-style window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays Python code and its output:

```
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> sorted(languages)
['C++', 'HTML', 'Java', 'Python']
>>> languages
['Python', 'C++', 'Java', 'HTML']
>>> |
```

In the bottom right corner of the window, there is a status bar with "Ln: 49 Col: 4".

Lists - Reverse

Another method that changes the original list is **.reverse()** :



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main area contains the following Python code:

```
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> languages.reverse()
>>> languages
['HTML', 'Java', 'C++', 'Python']
>>>
```

The status bar at the bottom right indicates "Ln: 101 Col: 33".

Questions?

INFORMATICS

Lecture 4

I210 – INTRODUCTION TO
PROGRAMMING WITH PYTHON

Today

- More List Methods
- Tuples
- Modules
- Script Mode
- The print function
- The input function

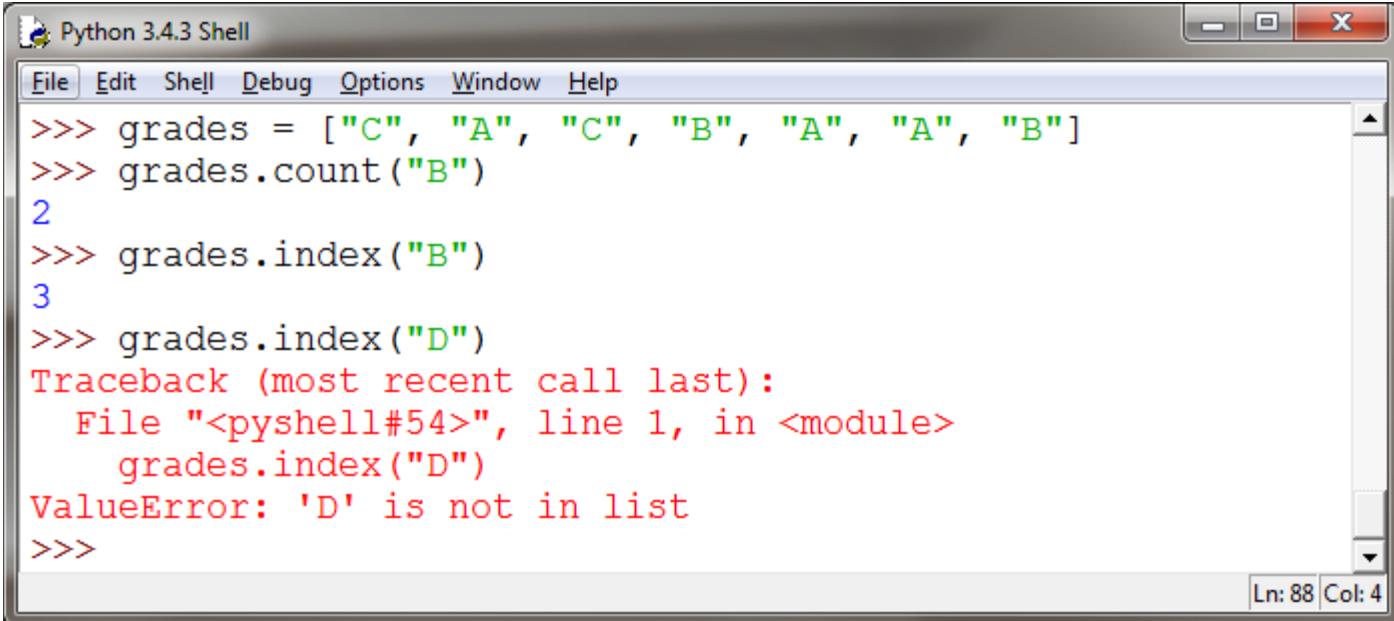
.join() .sort() copy() dir() help() input()

Lists – Count and Index

We can **count** occurrences of an element or find an **index** based on *value*.

This returns the index of the FIRST match only.

Caution! You'll get an error if your target isn't in the list!



The screenshot shows a Python 3.4.3 Shell window. The code entered is:

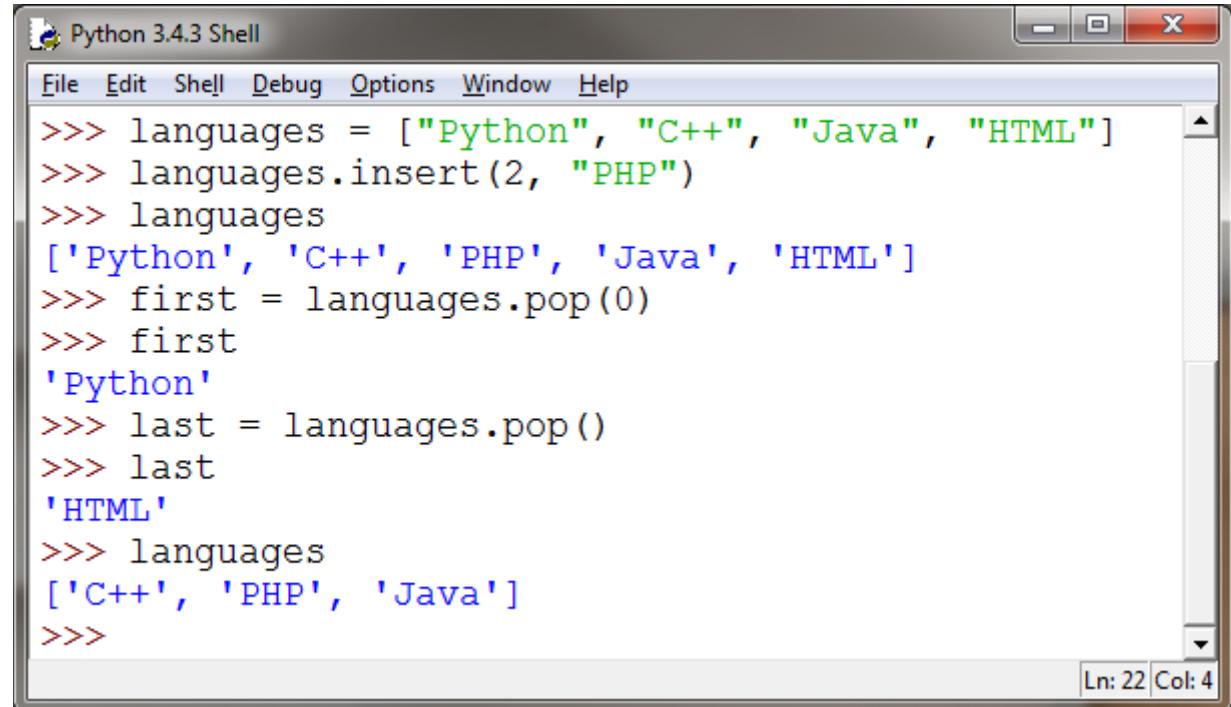
```
>>> grades = ["C", "A", "C", "B", "A", "A", "B"]
>>> grades.count("B")
2
>>> grades.index("B")
3
>>> grades.index("D")
Traceback (most recent call last):
  File "<pyshell#54>", line 1, in <module>
    grades.index("D")
ValueError: 'D' is not in list
>>>
```

The output shows the count of 'B' as 2 and its index as 3. However, when trying to find the index of 'D', it raises a `ValueError: 'D' is not in list`.

Lists – Inserting and Popping

We can insert elements at an index, and we can remove an element to use it.

- The default for **.pop()** is the last position if you don't pass it a number.



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> languages = ["Python", "C++", "Java", "HTML"]
>>> languages.insert(2, "PHP")
>>> languages
['Python', 'C++', 'PHP', 'Java', 'HTML']
>>> first = languages.pop(0)
>>> first
'Python'
>>> last = languages.pop()
>>> last
'HTML'
>>> languages
['C++', 'PHP', 'Java']
>>>
Ln: 22 Col: 4
```

Immutable vs Mutable Sequences

Mutable: Changeable once set at every level

Immutable: Unchangeable once set

- Unlike lists, strings are immutable sequences - they can't be altered at the element level.

The image shows two side-by-side Python 3.4.3 Shell windows. The left window shows a list mutation attempt that fails due to string immutability. The right window shows a successful list mutation.

Left Window (Failed Mutation):

```
>>> word = "gament"
>>> word[0]
Traceback (most recent call last):
  File "<pyshell>1", line 1, in <module>
    word[0]
TypeError: 'str' object does not support item assignment
```

Right Window (Successful Mutation):

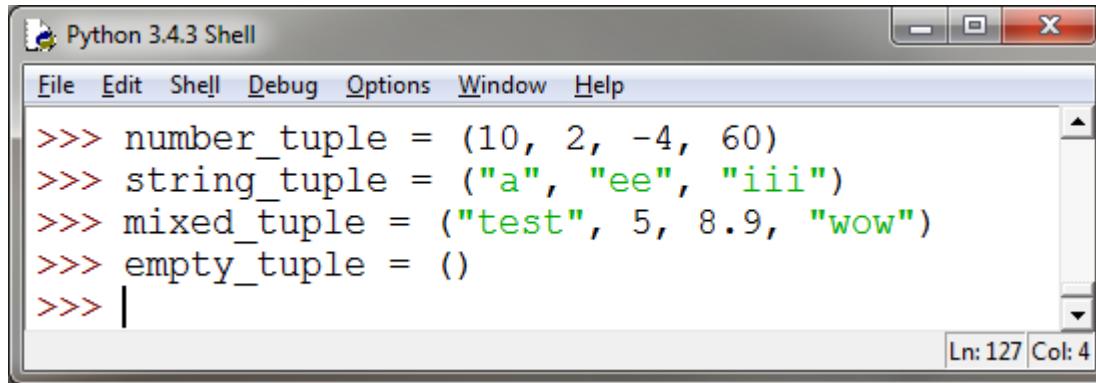
```
>>> letters = ["g", "a", "m", "e"]
>>> letters[0] = "s"
>>> letters
['s', 'a', 'm', 'e']
>>>
```

Tuples

Tuples are essentially immutable lists.

- (Tuples are also known as **arrays**.)

- We create tuples with parenthesis: ()



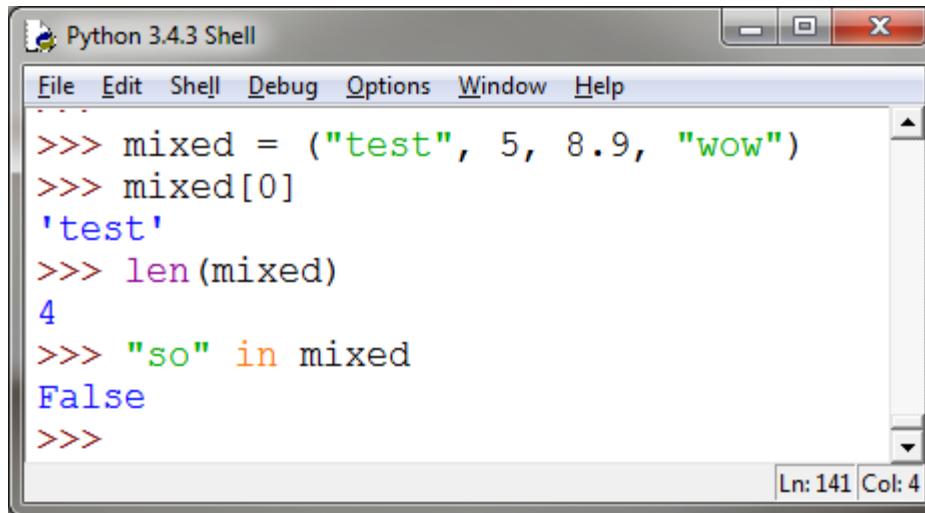
A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python code:

```
>>> number_tuple = (10, 2, -4, 60)
>>> string_tuple = ("a", "ee", "iii")
>>> mixed_tuple = ("test", 5, 8.9, "wow")
>>> empty_tuple = ()
>>> |
```

The status bar at the bottom right shows Ln: 127 Col: 4.

Tuples

In terms of indexing, **len**, and the **in** operator, tuples behave exactly like lists:



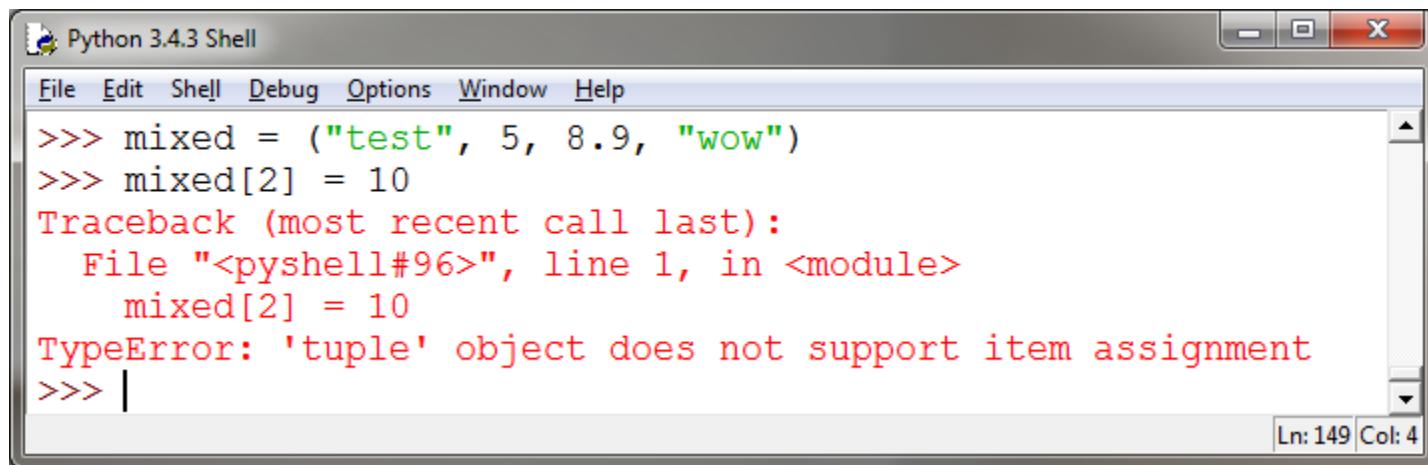
A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python session:

```
>>> mixed = ("test", 5, 8.9, "wow")
>>> mixed[0]
'test'
>>> len(mixed)
4
>>> "so" in mixed
False
>>>
```

The status bar at the bottom right shows Ln: 141 Col: 4.

Tuples

But tuples are **immutable**,
so we can't dynamically change their elements:



The screenshot shows a Windows-style window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays a Python session. The user has defined a tuple:

```
>>> mixed = ("test", 5, 8.9, "wow")
```

Then, they attempt to assign a value to the second element of the tuple:

```
>>> mixed[2] = 10
```

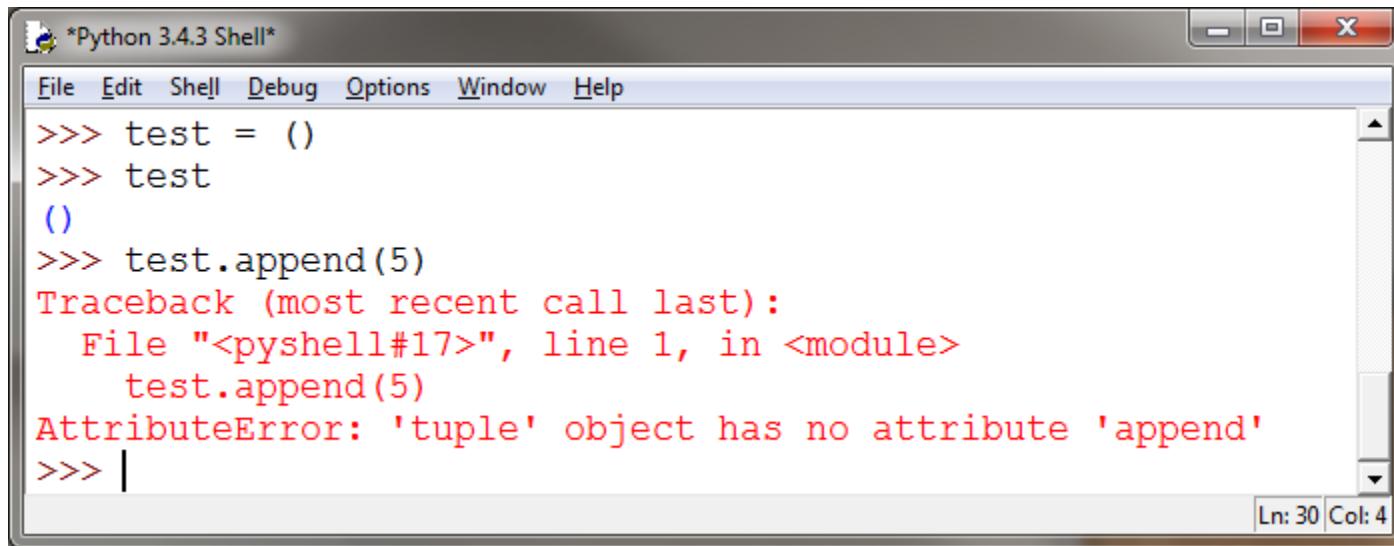
This results in a `TypeError`:

```
Traceback (most recent call last):
  File "<pyshell#96>", line 1, in <module>
    mixed[2] = 10
TypeError: 'tuple' object does not support item assignment
```

The bottom status bar indicates "Ln: 149 Col: 4".

Tuples

Tuples don't support a lot of the “fancier” options we have with lists, like **.append()**

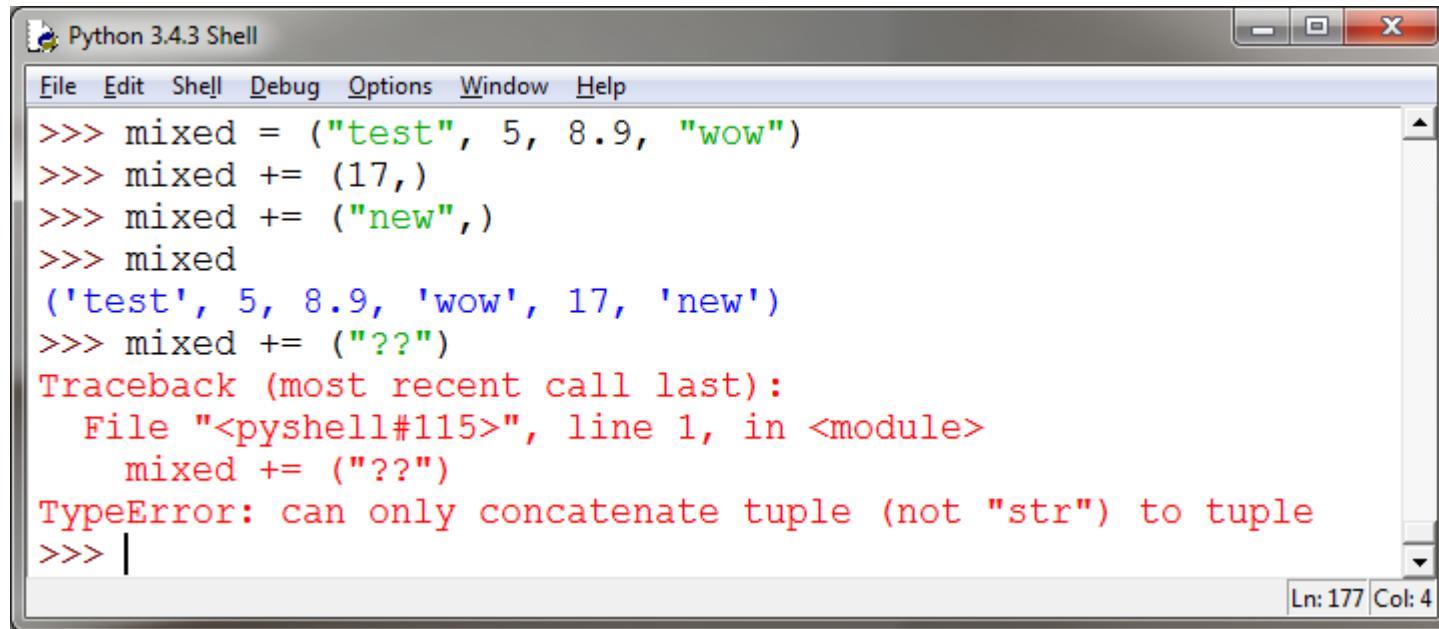


```
*Python 3.4.3 Shell*
File Edit Shell Debug Options Window Help
>>> test = ()
>>> test
()
>>> test.append(5)
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    test.append(5)
AttributeError: 'tuple' object has no attribute 'append'
>>> |
```

The screenshot shows a Python 3.4.3 Shell window. The user has created an empty tuple named 'test'. When they attempt to use the '.append()' method on it, they receive an 'AttributeError' indicating that tuples do not have an 'append' attribute.

Tuples

... but we can add elements to tuples with concatenation!



The screenshot shows a Python 3.4.3 Shell window. The user attempts to concatenate a tuple with a string, which results in a `TypeError`.

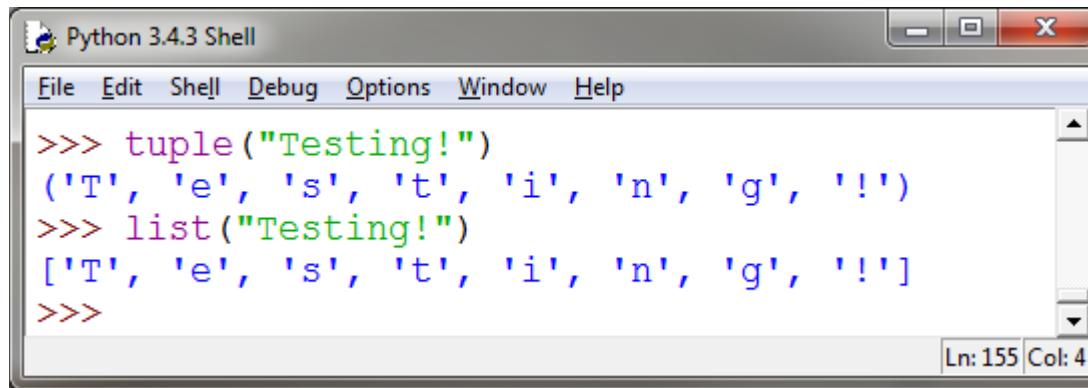
```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> mixed = ("test", 5, 8.9, "wow")
>>> mixed += (17,)
>>> mixed += ("new",)
>>> mixed
('test', 5, 8.9, 'wow', 17, 'new')
>>> mixed += ("??")
Traceback (most recent call last):
  File "<pyshell#115>", line 1, in <module>
    mixed += ("??")
TypeError: can only concatenate tuple (not "str") to tuple
>>> |
```

Ln: 177 Col: 4

Notice: if you don't put the **comma** there,
Python doesn't know it's a tuple, and you get *an error!*

The tuple() and list() Functions

There are **tuple()** and **list()** data conversion functions that take a string, but they create tuples and list of *characters*!



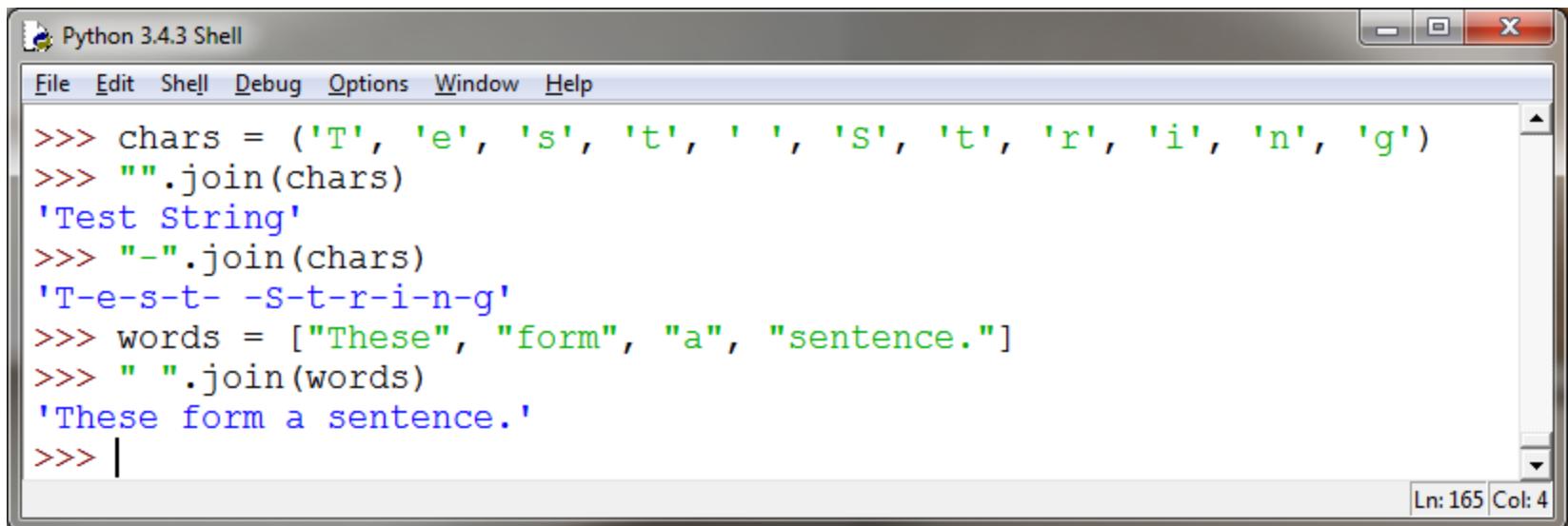
A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following code and output:

```
>>> tuple("Testing!")
('T', 'e', 's', 't', 'i', 'n', 'g', '!')
>>> list("Testing!")
['T', 'e', 's', 't', 'i', 'n', 'g', '!']
>>>
```

In the bottom right corner of the shell window, there is a status bar with "Ln: 155 Col: 4".

.join

The **.join()** method can reassemble a list or tuple of strings:



The screenshot shows a window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area contains the following Python code:

```
>>> chars = ('T', 'e', 's', 't', ' ', 's', 't', 'r', 'i', 'n', 'g')
>>> "".join(chars)
'Test String'
>>> "-".join(chars)
'T-e-s-t- -s-t-r-i-n-g'
>>> words = ["These", "form", "a", "sentence."]
>>> " ".join(words)
'These form a sentence.'
```

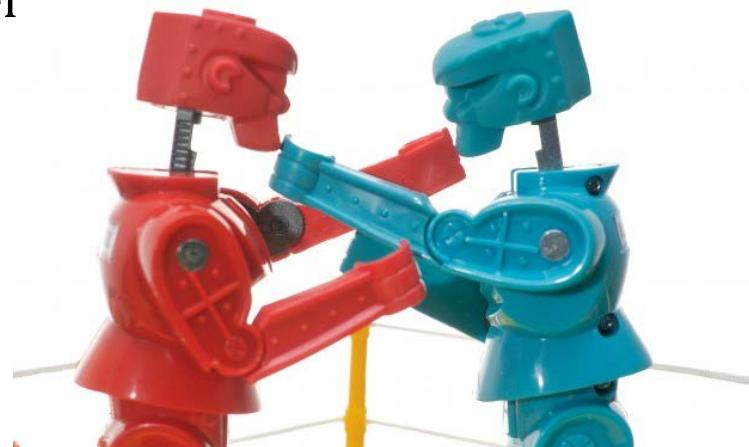
The output window shows the results of the code execution. The status bar at the bottom right indicates "Ln: 165 Col: 4".

Tuples vs. Lists – Why have both?

- Python can run code on tuples *faster* than on lists, because tuples have fewer features!
- Tuples are used for data that doesn't change much, like a sequence that stores the states in the USA.

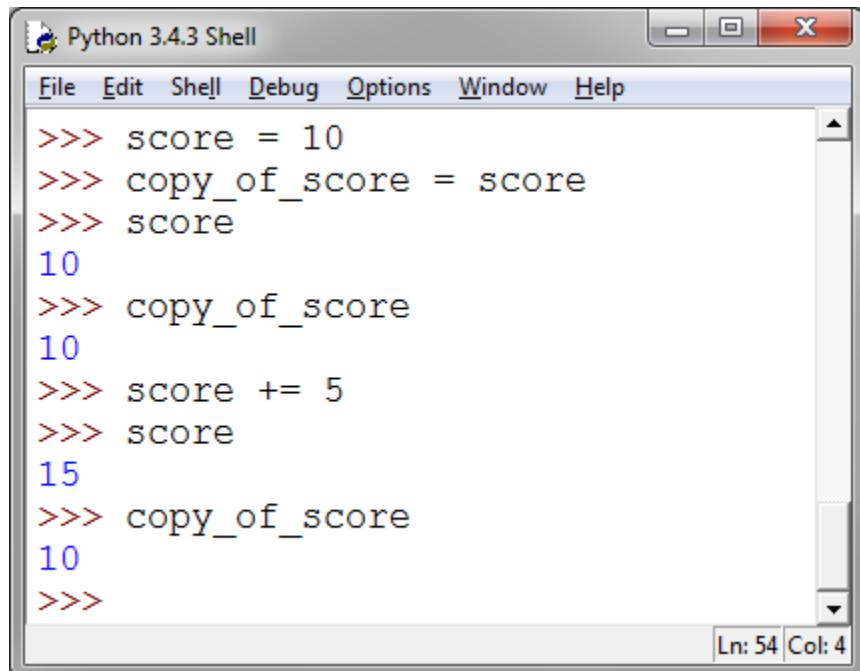
Classic trade-off in Programming: convenience vs. speed

- Assuming that elements can't change makes tuples much faster, but programming harder
- So Python gives you two options:
 - **tuples (fast but limited)**
 - **lists (slow but flexible)**



Shared References

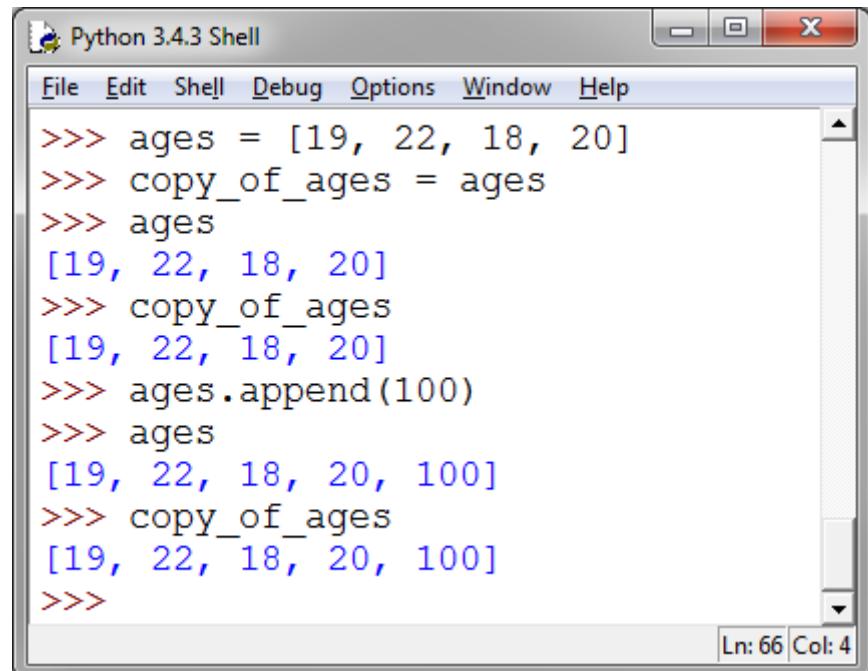
Take 1 minute to look at these two examples with your group. *Discuss what's different.*



Python 3.4.3 Shell

```
>>> score = 10
>>> copy_of_score = score
>>> score
10
>>> copy_of_score
10
>>> score += 5
>>> score
15
>>> copy_of_score
10
>>>
```

Ln: 54 Col: 4



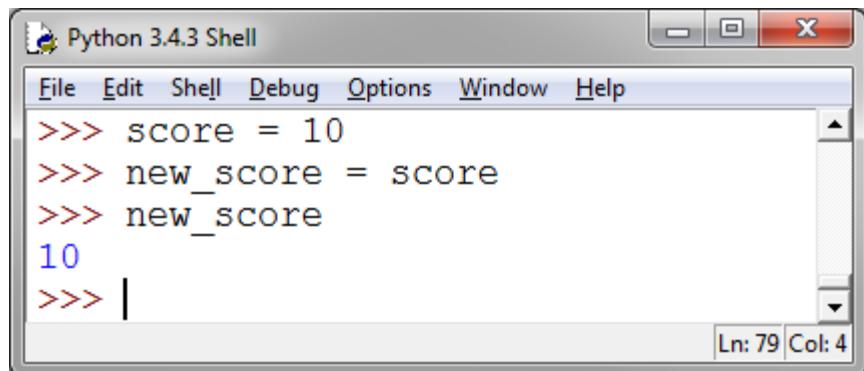
Python 3.4.3 Shell

```
>>> ages = [19, 22, 18, 20]
>>> copy_of_ages = ages
>>> ages
[19, 22, 18, 20]
>>> copy_of_ages
[19, 22, 18, 20]
>>> ages.append(100)
>>> ages
[19, 22, 18, 20, 100]
>>> copy_of_ages
[19, 22, 18, 20, 100]
>>>
```

Ln: 66 Col: 4

Shared References

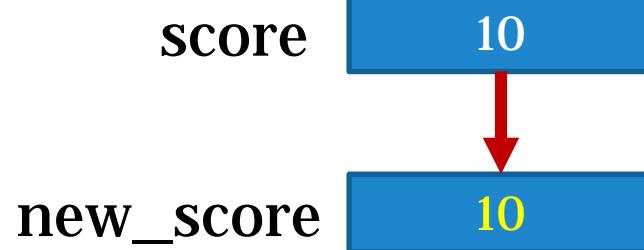
Simple values like ints, floats, and strings are *copied by value*. This means that when we assign one to a variable, the value is copied over.



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area shows:

```
>>> score = 10
>>> new_score = score
>>> new_score
10
>>> |
```

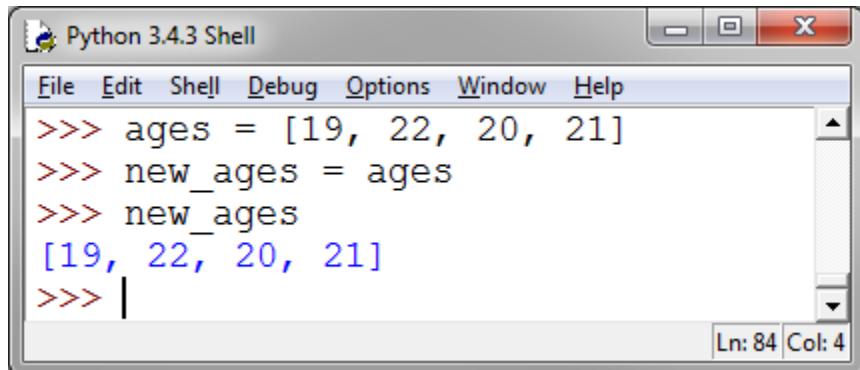
The status bar at the bottom right indicates Ln: 79 Col: 4.



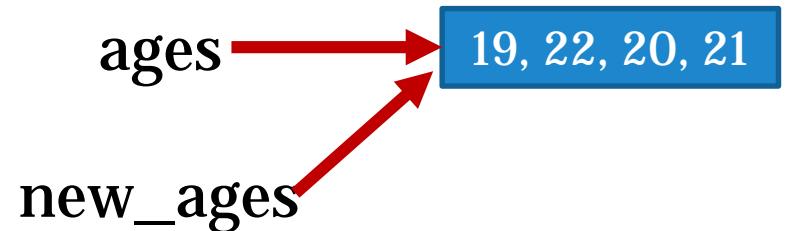
Shared References

Complex values like lists are *copied by reference*. That means that when we assign one to a variable, it makes a reference to the original.

So anything we do to one is done to the other!

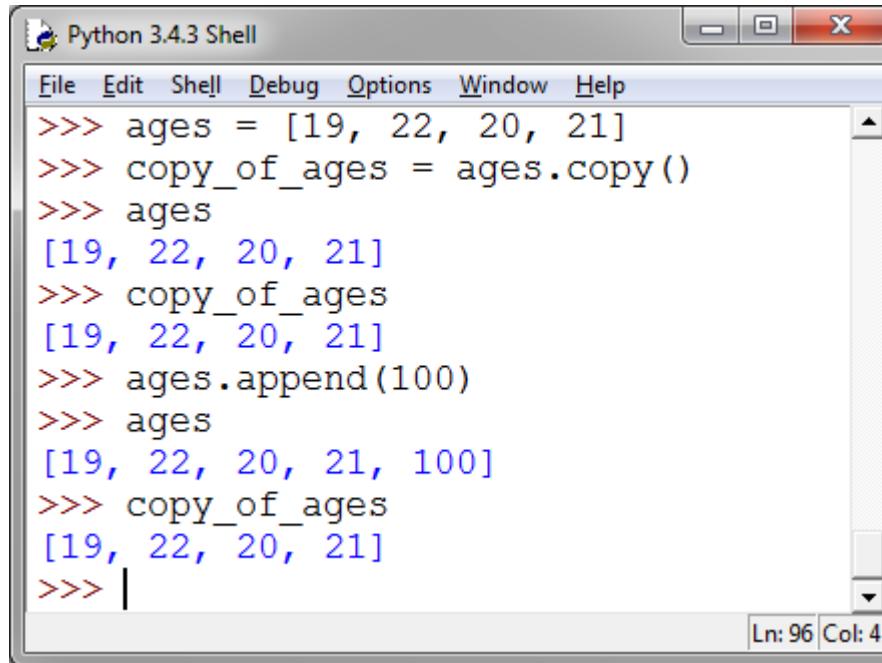


```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> ages = [19, 22, 20, 21]
>>> new_ages = ages
>>> new_ages
[19, 22, 20, 21]
>>> |
Ln: 84 Col: 4
```



The .copy() Method

We can create an actual copy with **.copy()** :



A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The code input area shows the following session:

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> ages = [19, 22, 20, 21]
>>> copy_of_ages = ages.copy()
>>> ages
[19, 22, 20, 21]
>>> copy_of_ages
[19, 22, 20, 21]
>>> ages.append(100)
>>> ages
[19, 22, 20, 21, 100]
>>> copy_of_ages
[19, 22, 20, 21]
>>> |
```

The status bar at the bottom right indicates Ln: 96 Col: 4.

Modules

Module: a file that contains a collection of programming code.

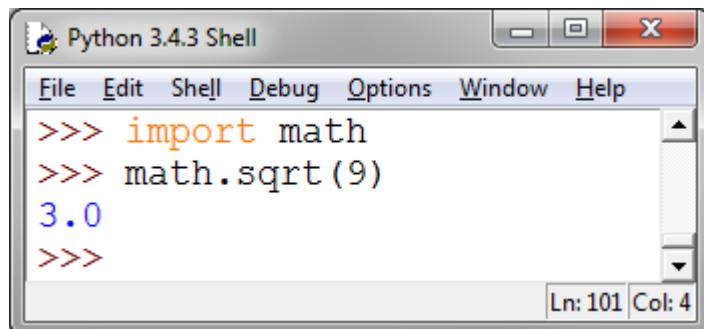
- Modules can be used by **importing** them, at which point the code becomes part of the current program.
- *Not all of Python is loaded by default!*
Importing allows us to use more of the language.

Using Modules

Modules contain groups of related functions:

- **math** contains a lot of useful mathematical code
- Modules can also contain classes or constants
 - **math.pi**

Normally when we import a module we then have to say **modulename.methodname()**



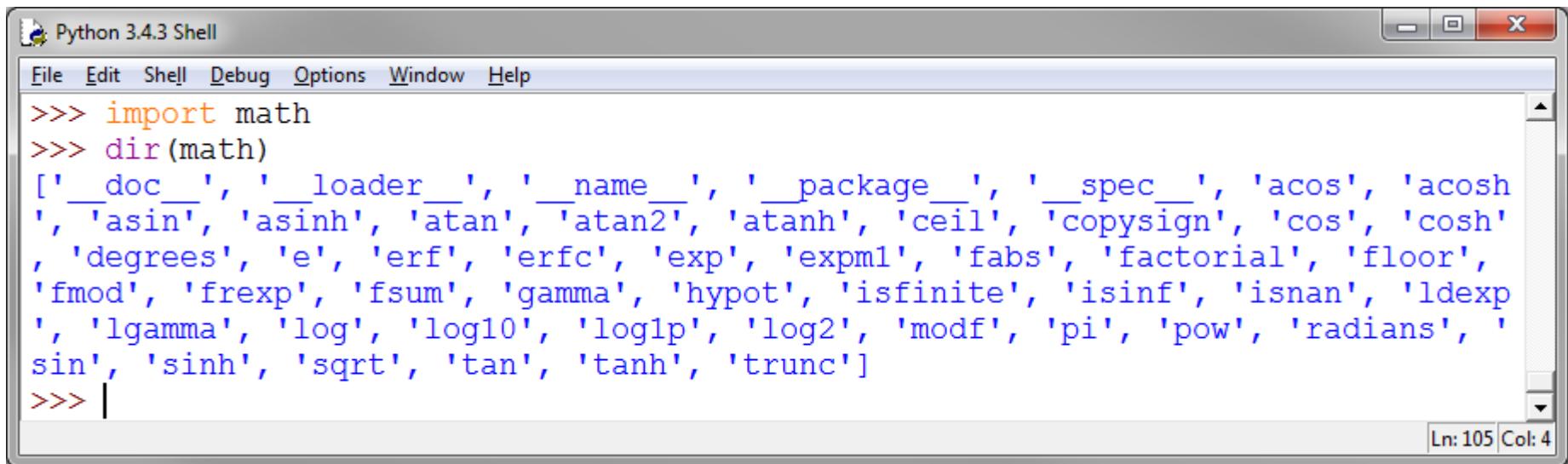
A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window shows the following command-line interaction:

```
>>> import math
>>> math.sqrt(9)
3.0
>>>
```

The status bar at the bottom right indicates "Ln: 101 Col: 4".

What's in a Module?

The built-in function **dir()** is used to find out **what a module defines**. It returns a sorted list of strings:



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window shows the following code and its output:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
 'acosh',
 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp',
 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians',
 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> |
```

In the bottom right corner of the shell window, there is a status bar with "Ln: 105 Col: 4".

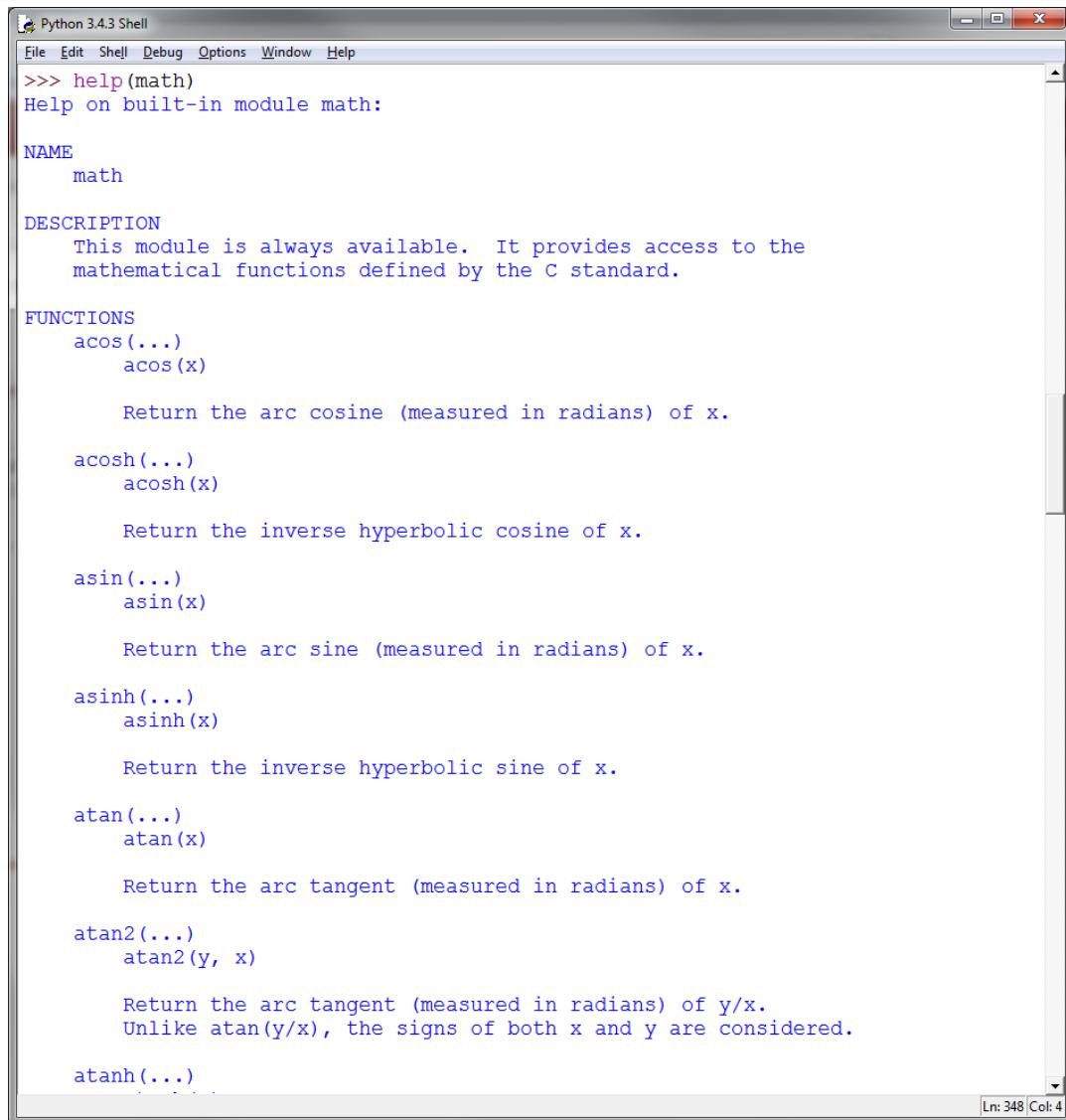
This can be helpful if you can't remember the name of a function in a module.

What's in a Module?

Need more?

Try the **help()**
function.

This gives a LOT
of information.



The screenshot shows a Windows-style window titled "Python 3.4.3 Shell". Inside, a command-line interface displays the output of the `help(math)` command. The output is as follows:

```
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    asinh(...)
        asinh(x)

        Return the inverse hyperbolic sine of x.

    atan(...)
        atan(x)

        Return the arc tangent (measured in radians) of x.

    atan2(...)
        atan2(y, x)

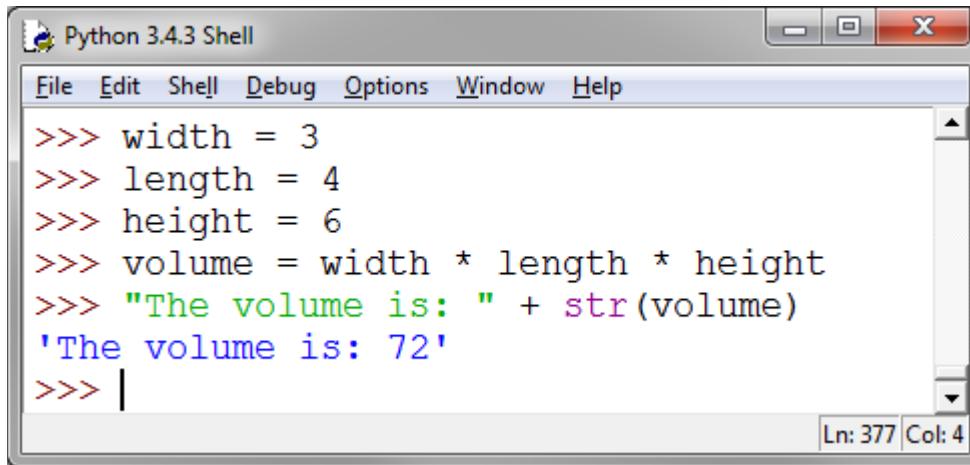
        Return the arc tangent (measured in radians) of y/x.
        Unlike atan(y/x), the signs of both x and y are considered.

    atanh(...)
```

The window has a title bar, menu bar, and status bar at the bottom right indicating "Ln: 348 Col: 4".

Volume (Example)

Here's a short program that calculates the volume of a rectangular solid that's 3 units wide, 4 units long, and 6 units high:



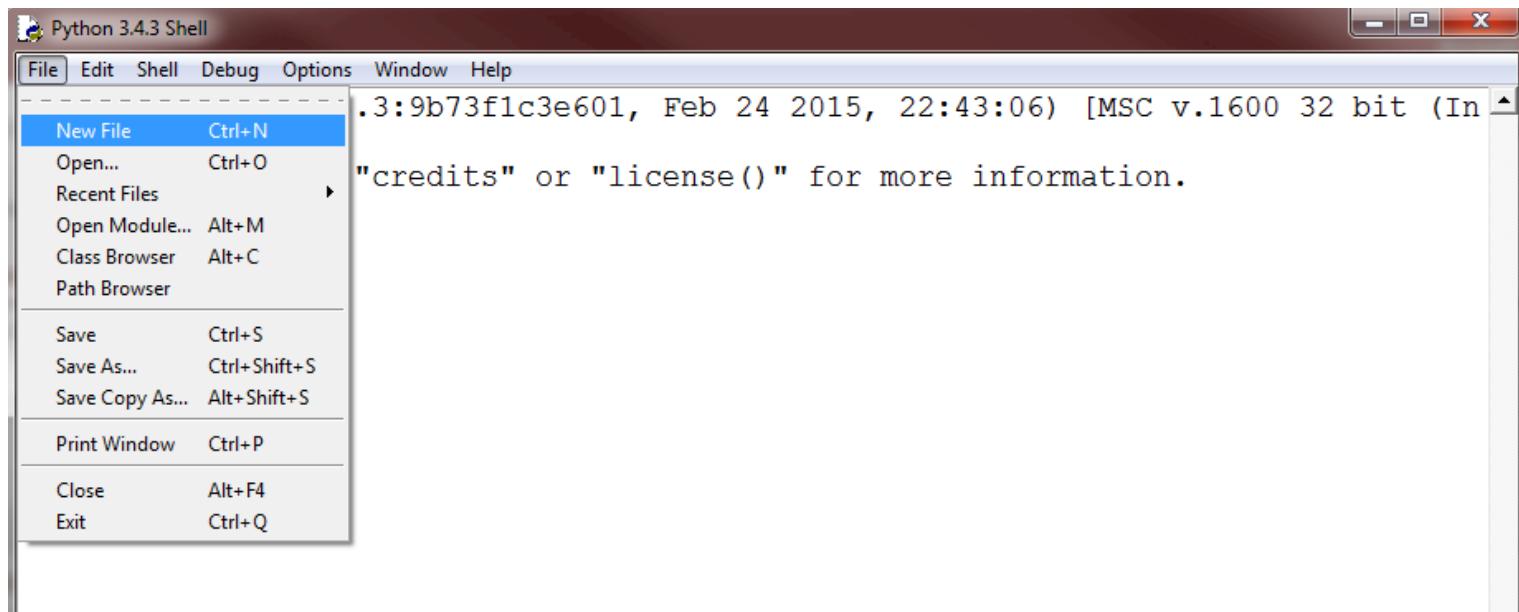
The screenshot shows a Windows-style window titled "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following Python code:

```
>>> width = 3
>>> length = 4
>>> height = 6
>>> volume = width * length * height
>>> "The volume is: " + str(volume)
'The volume is: 72'
>>> |
```

In the bottom right corner of the window, there is a status bar with "Ln: 377 Col: 4".

Script Mode

We open ***Script Mode*** in IDLE by choosing “New File”:

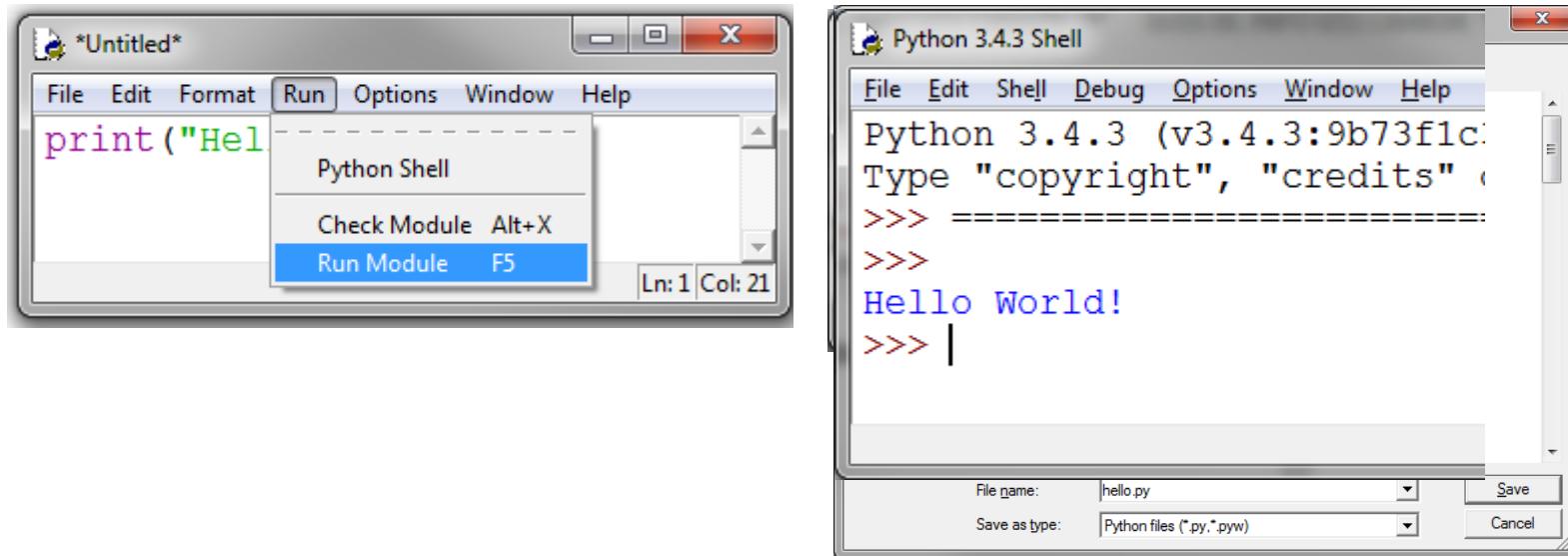


- This is where we'll write our programs from now on.
We can still use Interactive Mode to test things!

The print() Function

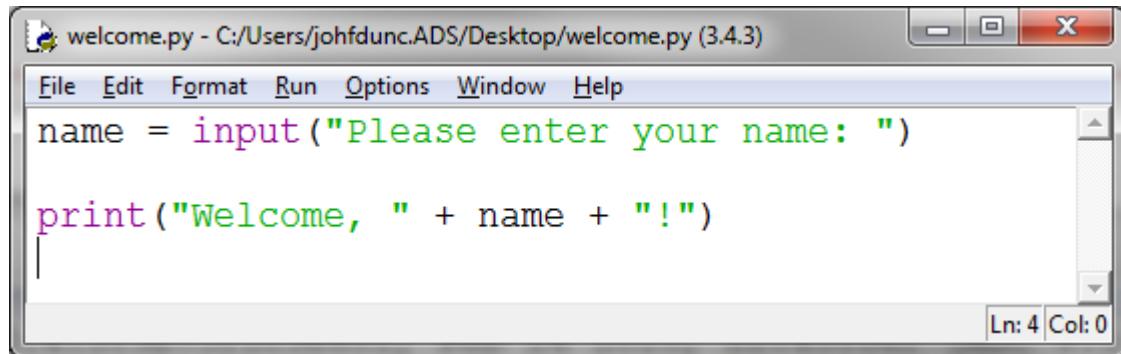
To see how *Script Mode* works, let's introduce a way to get **output** from a program.

- The **print()** function takes a string and outputs it to the console. Here's a traditional first program:



The `input()` Function

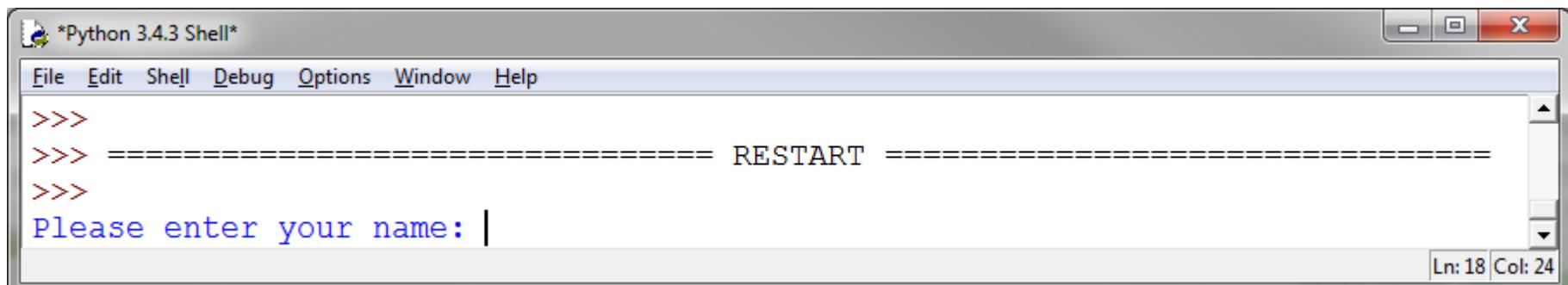
We also need a way to ask the user to input information:



```
welcome.py - C:/Users/johfdunc.ADS/Desktop/welcome.py (3.4.3)
File Edit Format Run Options Window Help
name = input("Please enter your name: ")

print("Welcome, " + name + "!")
Ln: 4 Col: 0
```

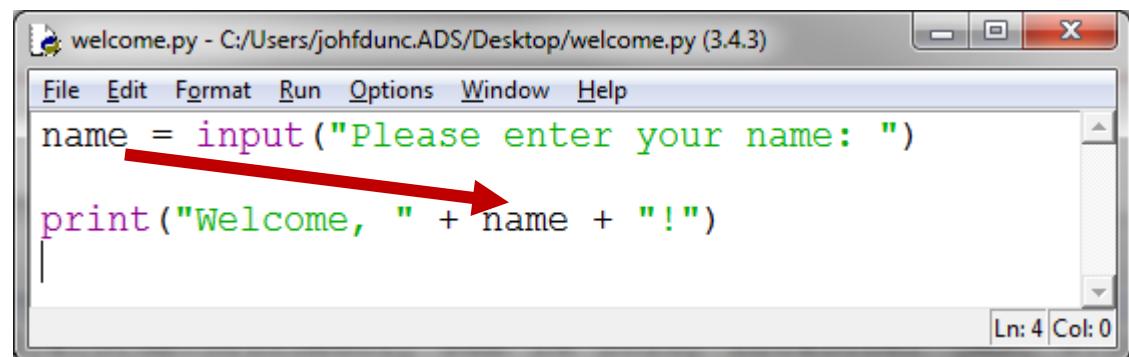
- The **input()** function prompts the user for keyboard input, and *returns a string*.



```
*Python 3.4.3 Shell*
File Edit Shell Debug Options Window Help
>>>
>>> ===== RESTART =====
>>>
Please enter your name: |
Ln: 18 Col: 24
```

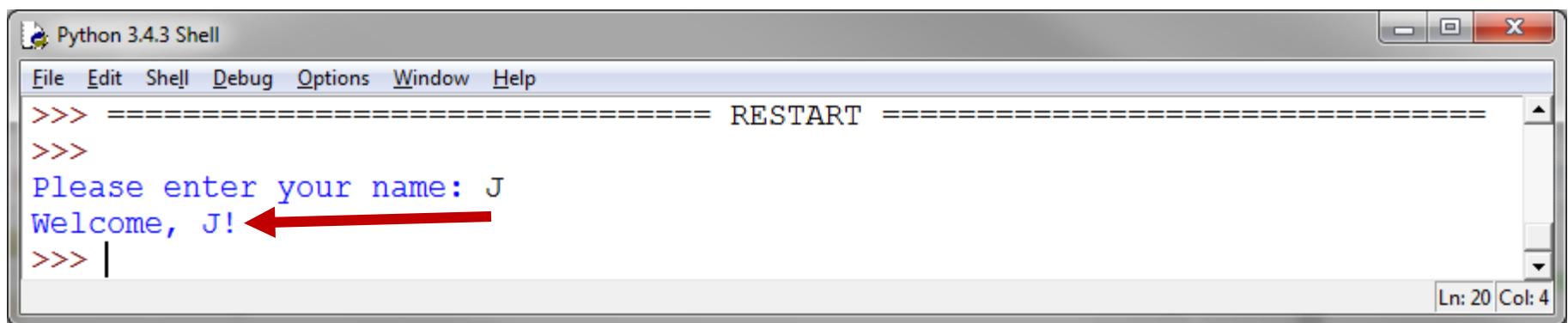
The input() Function

We can use information the user gives us in our programs!



```
welcome.py - C:/Users/johfdunc.ADS/Desktop/welcome.py (3.4.3)
File Edit Format Run Options Window Help
name = input("Please enter your name: ")
print("Welcome, " + name + "!")


Ln: 4 Col: 0
```

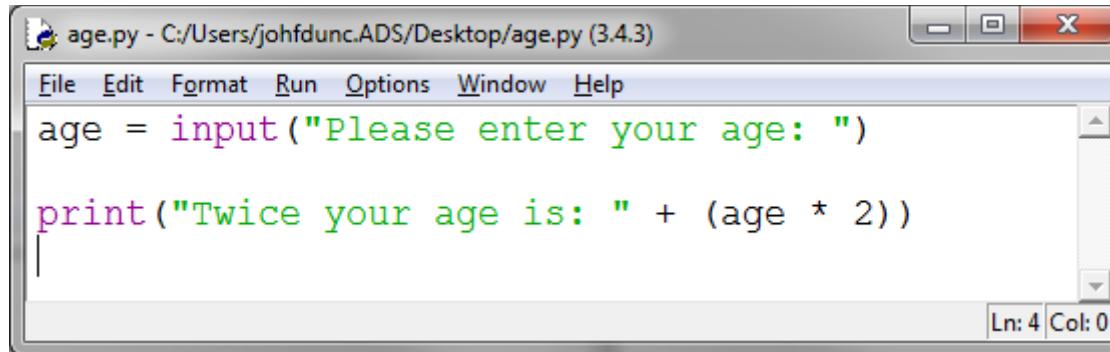


```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> ===== RESTART =====
>>>
Please enter your name: J
Welcome, J!
>>> |


Ln: 20 Col: 4
```

Using Data From the User

What went wrong here?

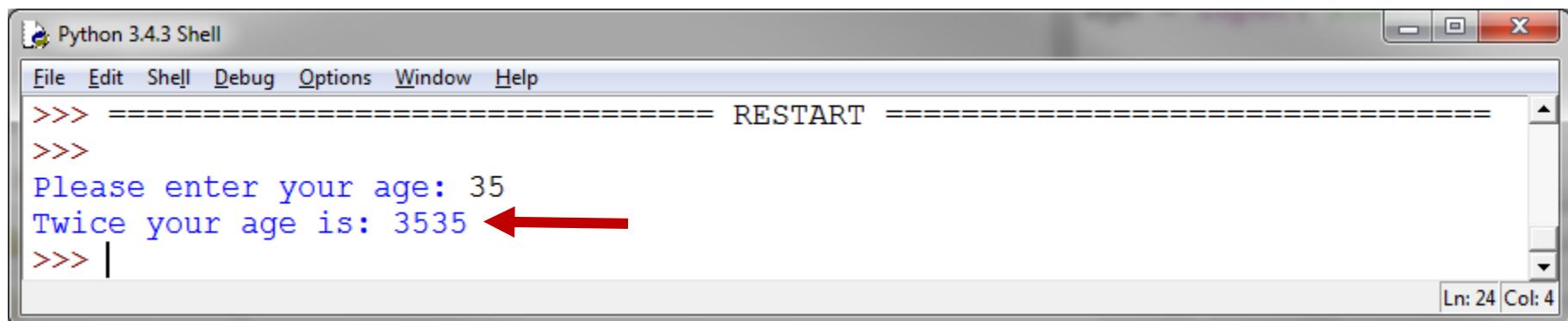


age.py - C:/Users/johfdunc.ADS/Desktop/age.py (3.4.3)

```
File Edit Format Run Options Window Help
age = input("Please enter your age: ")

print("Twice your age is: " + (age * 2))
```

Ln: 4 Col: 0



Python 3.4.3 Shell

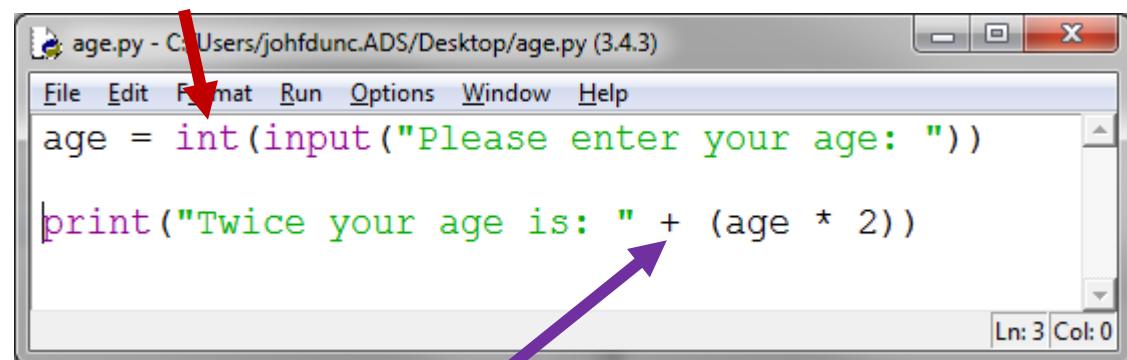
```
File Edit Shell Debug Options Window Help
>>> ===== RESTART =====
>>>
Please enter your age: 35
Twice your age is: 3535 ←
>>> |
```

Ln: 24 Col: 4

Using Data From the User

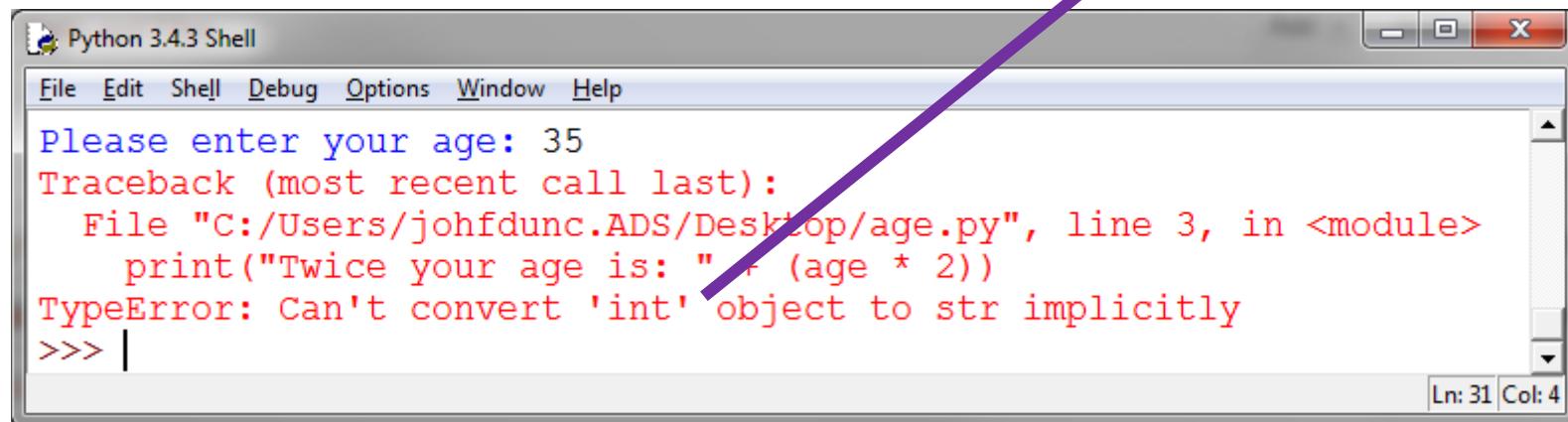
We probably need to convert the data from the user to an integer.

But that causes this problem:



```
age.py - C:\Users\johfdunc.ADS\Desktop\age.py (3.4.3)
File Edit Format Run Options Window Help
age = int(input("Please enter your age: "))

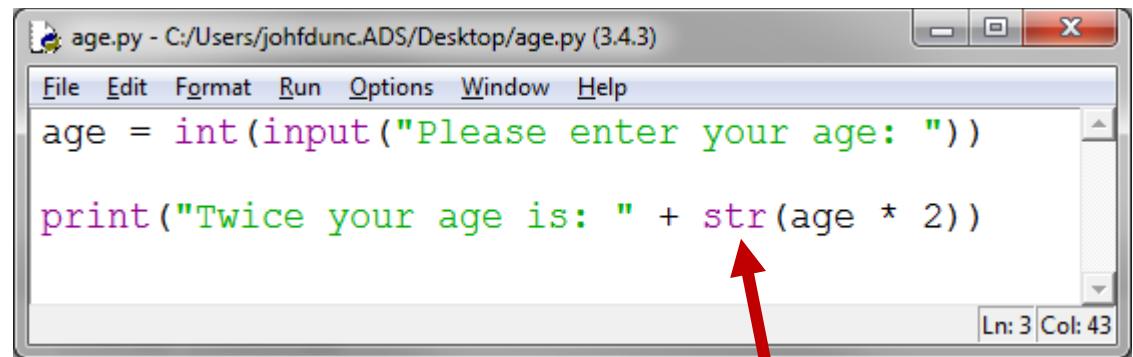
print("Twice your age is: " + (age * 2))
Ln: 3 Col: 0
```



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
Please enter your age: 35
Traceback (most recent call last):
  File "C:/Users/johfdunc.ADS/Desktop/age.py", line 3, in <module>
    print("Twice your age is: " + (age * 2))
TypeError: Can't convert 'int' object to str implicitly
>>> |
Ln: 31 Col: 4
```

Using Data From the User

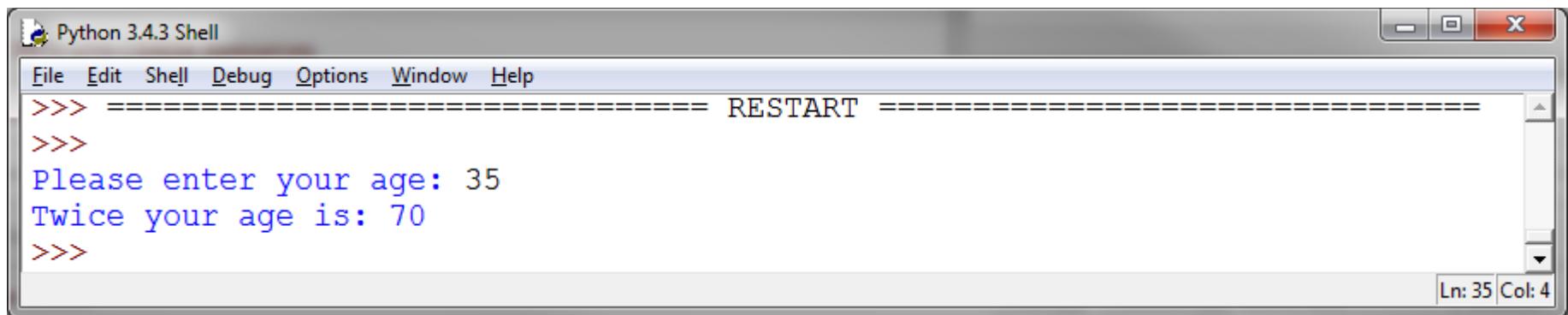
So we have to convert it BACK when we print!



```
age.py - C:/Users/johfdunc.ADS/Desktop/age.py (3.4.3)
File Edit Format Run Options Window Help
age = int(input("Please enter your age: "))

print("Twice your age is: " + str(age * 2))
```

Now it works!



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> ===== RESTART =====
>>>
Please enter your age: 35
Twice your age is: 70
>>>
```

Comments

- Are a reminder, any line that starts with a `#` will be ignored by Python. IDLE will highlight it in red.
- Comments are very important tools for organizing code and communicating with other people who may need to read it!
- Comments are a factor in grading.
- It's not 1 comment per 1 line. Use comments to organize sections of the code.

Questions?

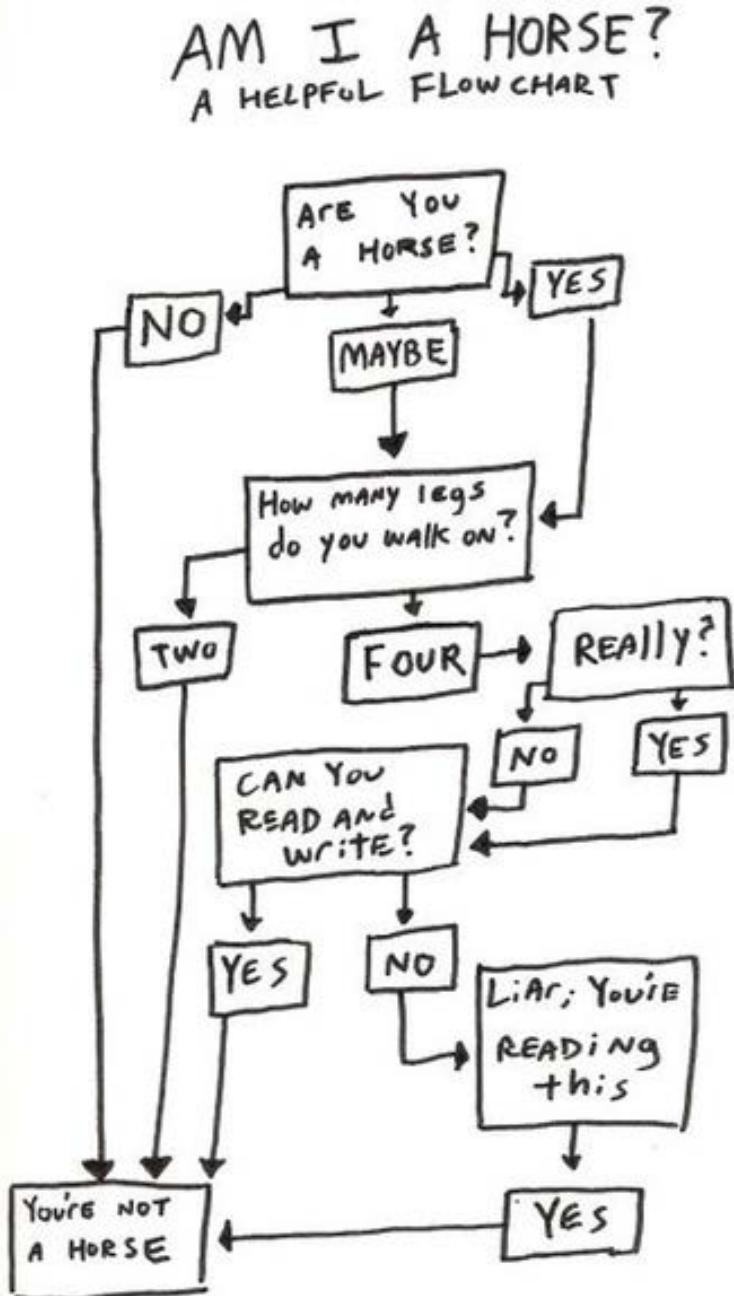
INFORMATICS

Lecture 5

I210 – INTRODUCTION TO
PROGRAMMING WITH PYTHON

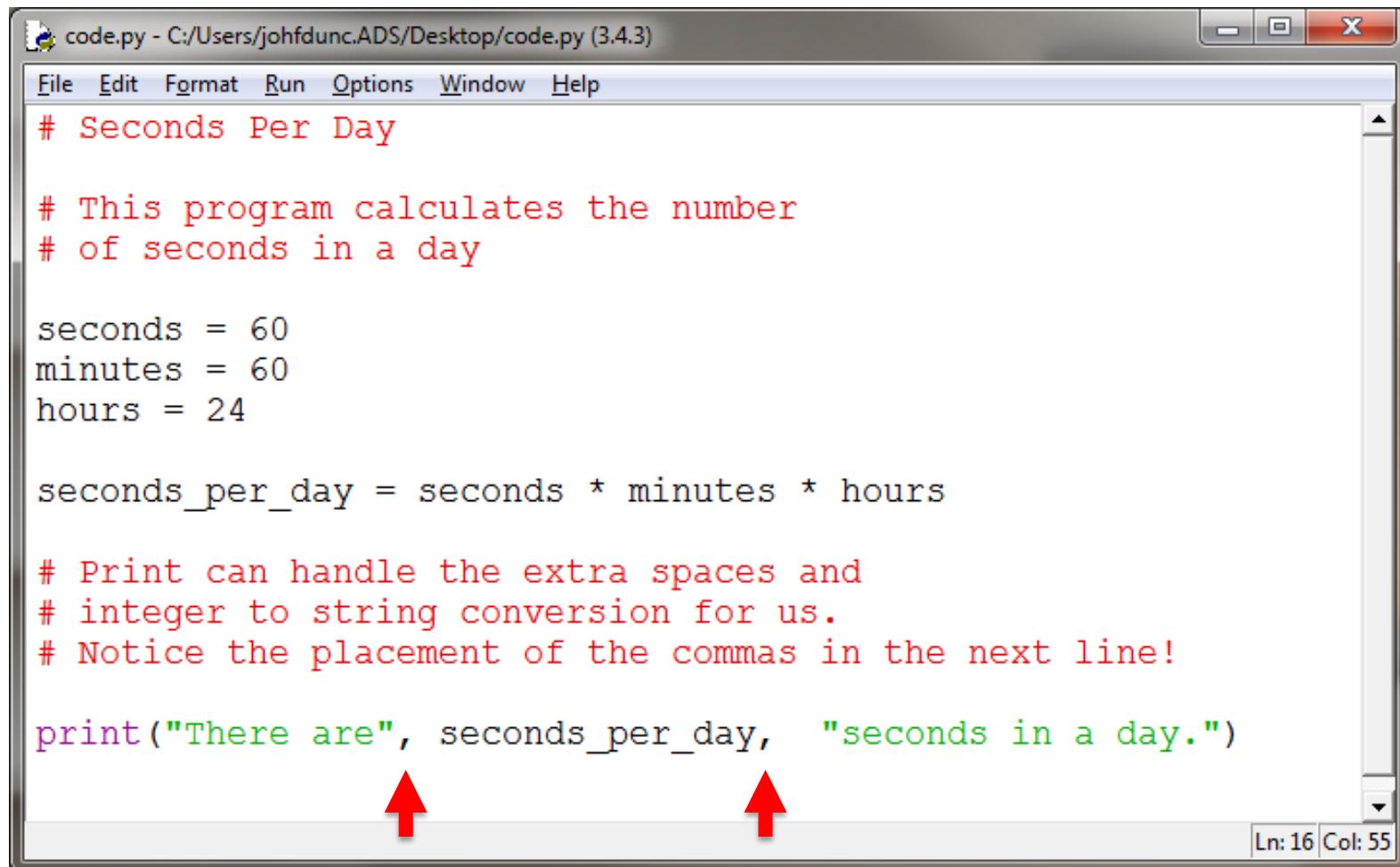
Today

- Comments
- The `eval()` function
- Making decisions:
 - If statements
 - Else clauses



print() with commas

The **print()** function has a shortcut with commas.



```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
# Seconds Per Day

# This program calculates the number
# of seconds in a day

seconds = 60
minutes = 60
hours = 24

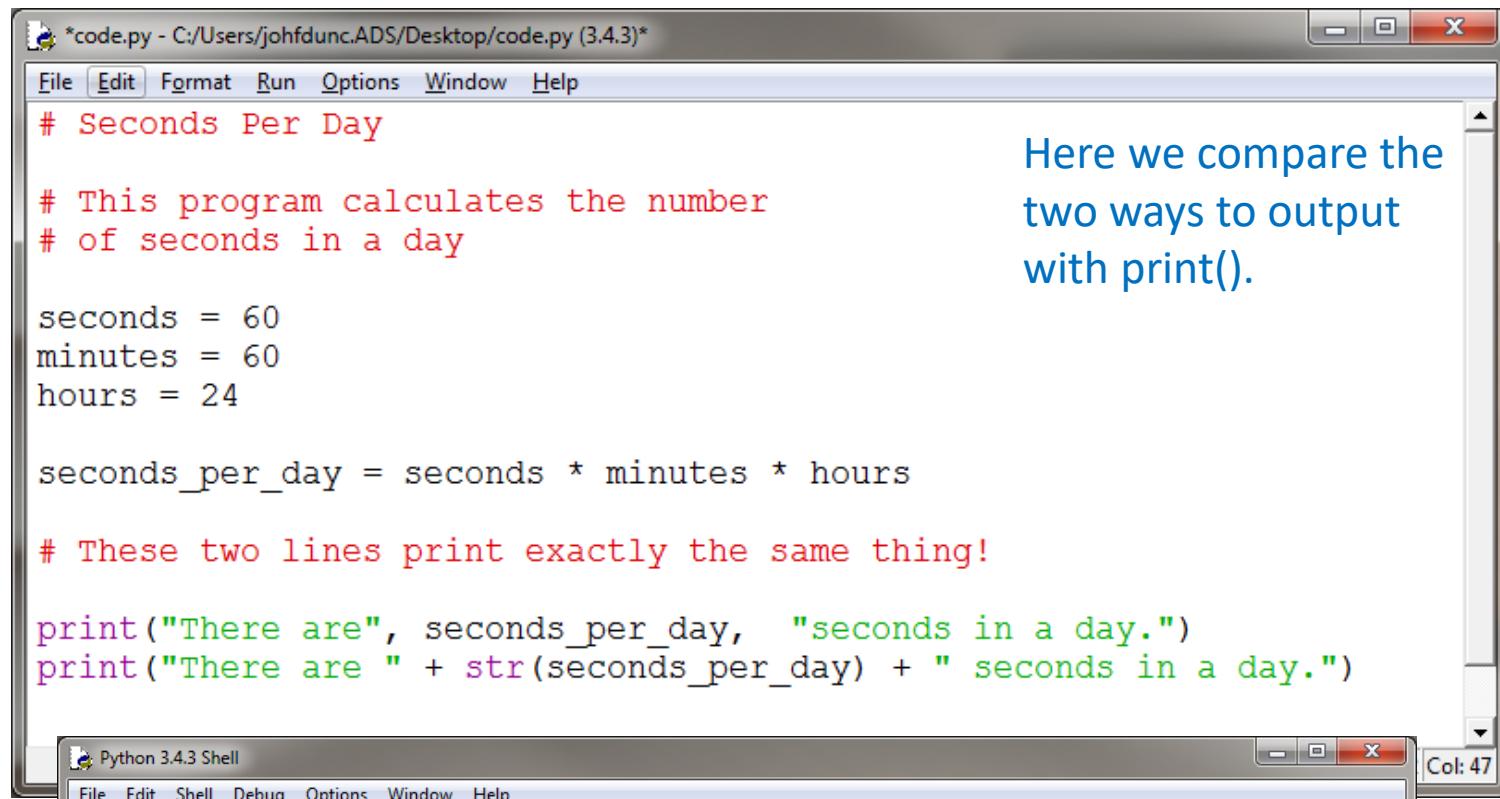
seconds_per_day = seconds * minutes * hours

# Print can handle the extra spaces and
# integer to string conversion for us.
# Notice the placement of the commas in the next line!

print("There are", seconds_per_day, "seconds in a day.")

Ln: 16 Col: 55
```

Seconds Per Day (Version 2)



```
*code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)*
File Edit Format Run Options Window Help
# Seconds Per Day

# This program calculates the number
# of seconds in a day

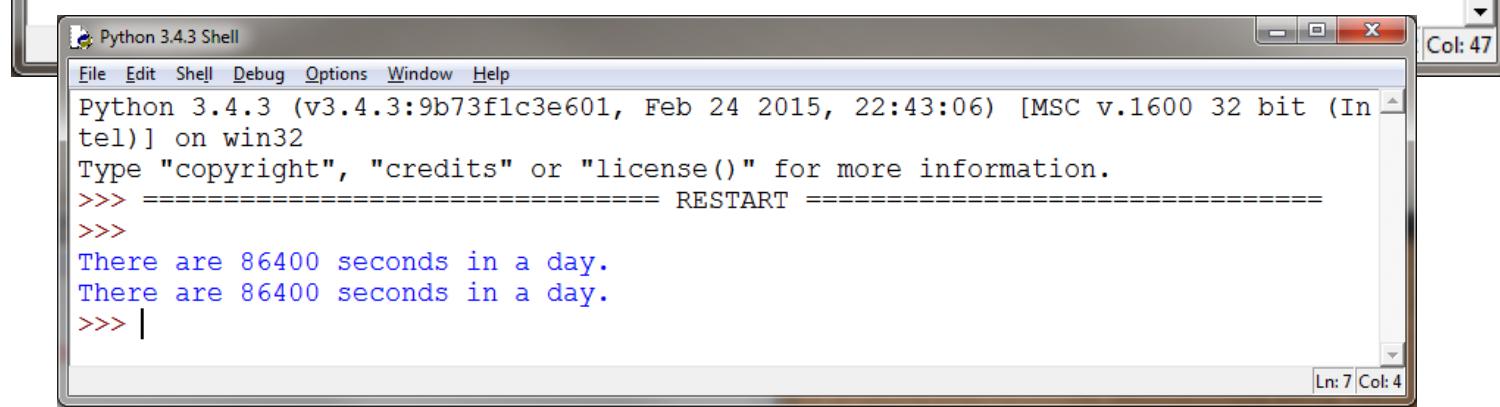
seconds = 60
minutes = 60
hours = 24

seconds_per_day = seconds * minutes * hours

# These two lines print exactly the same thing!

print("There are", seconds_per_day, "seconds in a day.")
print("There are " + str(seconds_per_day) + " seconds in a day.")
```

Here we compare the two ways to output with `print()`.

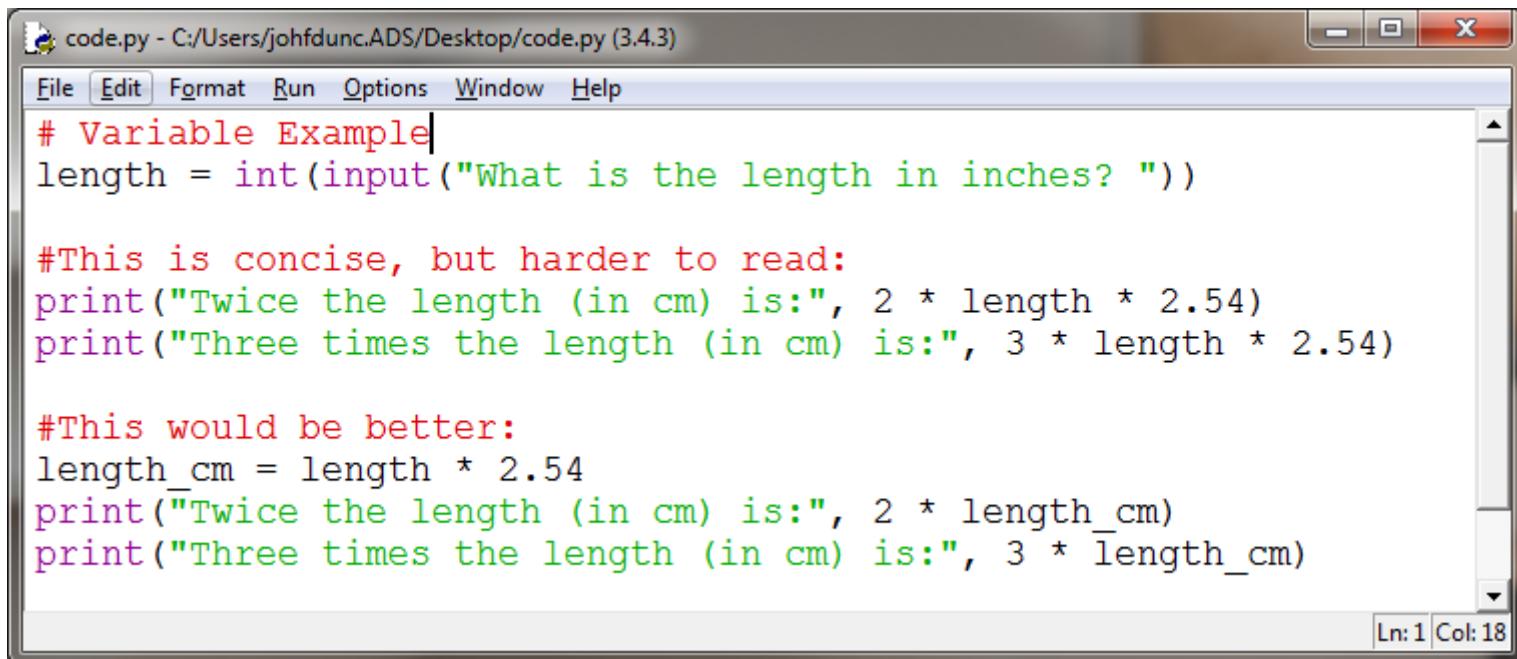


```
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
There are 86400 seconds in a day.
There are 86400 seconds in a day.
>>> |
```

Style Tip – Variable Use

Use variables when:

1. The same result is needed in more than one place.
2. Naming the information makes the code more legible.



The screenshot shows a Windows-style application window titled "code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The main text area contains the following Python code, demonstrating variable use:

```
# Variable Example
length = int(input("What is the length in inches? "))

#This is concise, but harder to read:
print("Twice the length (in cm) is:", 2 * length * 2.54)
print("Three times the length (in cm) is:", 3 * length * 2.54)

#This would be better:
length_cm = length * 2.54
print("Twice the length (in cm) is:", 2 * length_cm)
print("Three times the length (in cm) is:", 3 * length_cm)
```

The status bar at the bottom right indicates "Ln: 1 Col: 18".

The eval() function

Need to get something other than a string from the user?
Try the **eval()** function! Python *evaluates* the expression.

The image shows a Windows desktop environment with two open windows. The top window is titled "code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)" and contains Python code demonstrating the eval() function. The bottom window is titled "Python 3.4.3 Shell" and shows the execution of this code, prompting the user for input and displaying the results.

```
# eval() Example

year = eval(input("What year is it? "))
print("10 years from now is", year + 10, ".") 

numbers = eval(input("Please enter a list of numbers ([x,y,z]): "))
print("The sum of the numbers is:", sum(numbers), ".") 

letters = eval(input("Please enter a tuple of letters ('a', 'b', etc): "))
print("The first letter is:", letters[0], ".")
```

```
>>> ===== RESTART =====
>>>
What year is it? 2015
10 years from now is 2025 .
Please enter a list of numbers ([x,y,z]): [1,2,3,4]
The sum of the numbers is: 10 .
Please enter a tuple of letters ('a', 'b', etc): ('x', 'y', 'z')
The first letter is: x .
>>>
```

Ln: 15 Col: 4

A Runtime Error Waiting to Happen

eval() won't save you from the user entering a zero value!

```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
# Watch out!

distance = eval(input("How far did you go? "))
time = eval(input("How long did it take you? "))

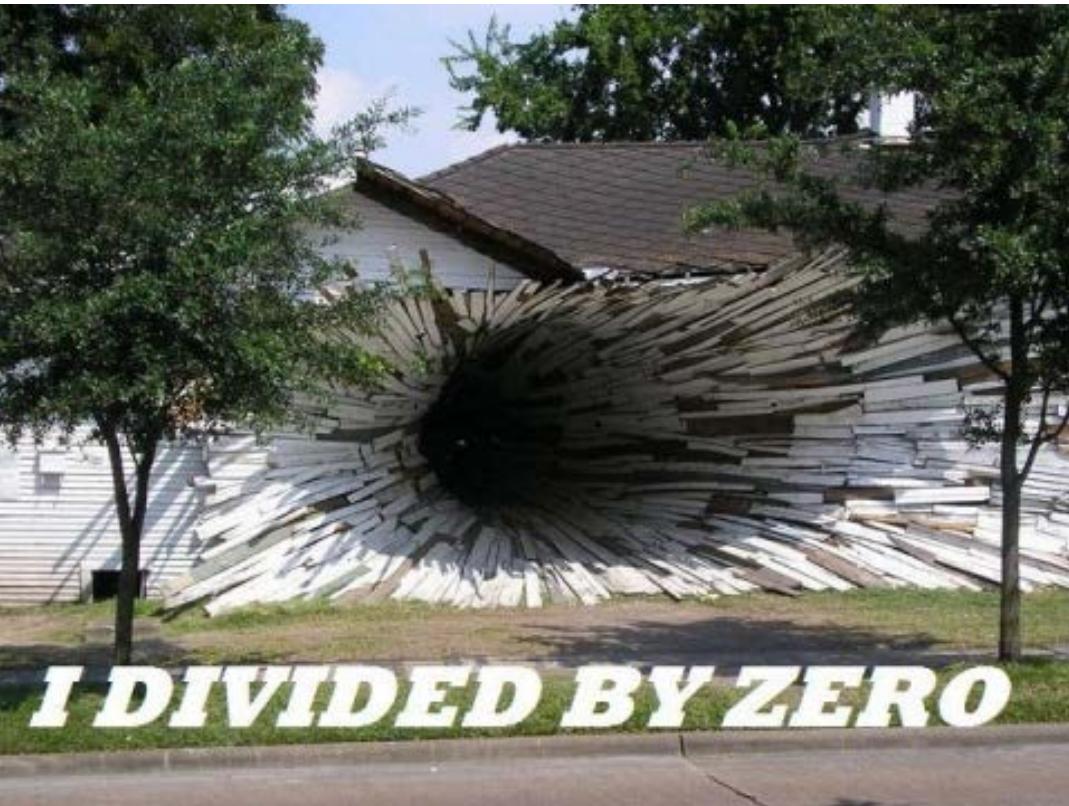
print("Your average speed was ", distance/time, ".")
```

Ln: 4 Col: 43

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>> ===== RESTART =====
>>>
How far did you go? 5
How long did it take you? 0
Traceback (most recent call last):
  File "C:/Users/johfdunc.ADS/Desktop/code.py", line 6, in <module>
    print("Your average speed was ", distance/time, ".")
ZeroDivisionError: division by zero
>>> |
```

Ln: 37 Col: 4

Always Watch Out for Zero Division!



How can we guard our programs (and houses) from the terror of **division by 0**?

Our programs need to be able to make decisions...

Short Greeting Program

A screenshot of a Windows-style application window titled "code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code editor contains the following Python script:

```
# Get the user's name
name = input("Please enter your name: ")

# Get the user's age
age = eval(input("Please enter your age: "))

# Display a welcome message
print("Thanks for using this program, " + name + ".")
print("Nice to meet a " + str(age) + "-year old!")
```

The status bar at the bottom right shows "Ln: 4 Col: 0".

A screenshot of the Python 3.4.3 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The shell prompt shows the text "RESTART". The user interaction and program output are as follows:

```
>>> ===== RESTART =====
>>>
Please enter your name: Abby
Please enter your age: 24
Thanks for using this program, Abby.
Nice to meet a 24-year old!
>>>
```

The status bar at the bottom right shows "Ln: 52 Col: 0".

Branching

We ask for the user's age...

- Special greeting for our seniors (65+)?



If the user is 65 or older, output:

“Don’t forget to ask about our senior discounts!”

We'll use a **Boolean Condition** and an **if statement**.

If statement

if Boolean Condition:

#Whatever we want to happen if the condition is True

#Something else we want to happen

#Something that needs to happen all the time

- Indent the lines of code that go under the **if** statement.
 - You can use TAB (*recommended*) or SPACE (2-4 spaces)

If statement



```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
# Our first branch in a program!

# Get the user's name
name = input("Please enter your name: ")

# Get the user's age
age = eval(input("Please enter your age: "))

# Display a welcome message
print("Thanks for using this program, ", name + ".")
print("Nice to meet a " + str(age) + "-year old!")

#check to see if we have a senior citizen
if age >= 65:
    print("Don't forget to ask about our senior discounts!")|
```

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>>
Please enter your name: Dave
Please enter your age: 72
Thanks for using this program, Dave.
Nice to meet a 72-year old!
Don't forget to ask about our senior discounts!
>>> |
```

Ln: 60 Col: 4

Branching – if and else

What if we want to do something if the Boolean Condition evaluates to False?

- We can use an OPTIONAL statement, **else**.

if Boolean Condition:

#Whatever we want to happen if this condition is True

else: *#NOTE – no Boolean Condition!*

#This will happen if the else clause is False

Branching - if and else

```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
# An if with an optional else clause!

# Get the user's age
age = eval(input("Please enter your age: "))

# Check to see if we have a senior citizen
# Younger people get a different message.
if age >= 65:
    print("Don't forget to ask about our senior discounts!")
else:
    print("We hope you shop here until you're old!")

Ln: 4 Col: 38
```

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>>
Please enter your age: 19
We hope you shop here until you're old!
>>> ===== RESTART =====
>>>
Please enter your age: 68
Don't forget to ask about our senior discounts!
>>> |
Ln: 76 Col: 4
```

ATM Program

```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
#A simple program for an ATM that dispenses $10 bills

money = eval(input("Welcome to the ATM. Please enter an amount to withdraw: "))

if money % 10 == 0:          #this means 'evenly divisibly by 10'
    print("You have withdrawn $", money)
else :                      #notice, no Boolean condition here!!
    print("I'm sorry. We cannot dispense that amount.")

print("Thanks for using the ATM.")

Ln: 8 Col: 55
```

```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>>
Welcome to the ATM. Please enter an amount to withdraw: 50
You have withdrawn $ 50
Thanks for using the ATM.
>>> ===== RESTART =====
>>>
Welcome to the ATM. Please enter an amount to withdraw: 34
I'm sorry. We cannot dispense that amount.
Thanks for using the ATM.
>>> |
Ln: 86 Col: 4
```

Indentation and Code Blocks

Correct:

```
if password == "secret":  
    print("Access Granted")  
else:  
    print("Access Denied")
```

Incorrect:

```
if password == "secret":  
    print("Access Granted")  
else:  
    print("Access Denied")
```

Indenting (or using spaces) creates **code blocks**
– units of code with a meaning!

Treating Values as Conditions

Any value can be interpreted as **True** or **False** (as a Boolean condition)!

- Any empty (**None**) or zero value is **False**
 - So, **0**, **" "**, and **None** are **False**
- Any other non-empty value is **True**
 - So for example, **-10**, **2.5**, **"banana"** are **True**

`if money:`

- **money** is treated as a Boolean condition
- **True** when **money** is not **0**, **" "**, or **None**
- **False** when **money** is **0**, **" "**, or **None**

Branching

- What if we want to make multiple decisions?

“Ask a user to vote for Candidate A, B, or C, and display a different message for each, plus a message if they try to ‘write in’ someone else.”
- We could do this with multiple if statements, but there’s a better way – the **elif statement** (short for ‘*else if*’).

Like else, elif is an optional "addition" to an if statement.

If, elif, and else statements

if Boolean Condition:

#Whatever we want to happen if this condition is true

elif Other Boolean Condition:

#Whatever we want to happen if this condition is true

elif Third Boolean Condition:

#Whatever we want to happen if this condition is true

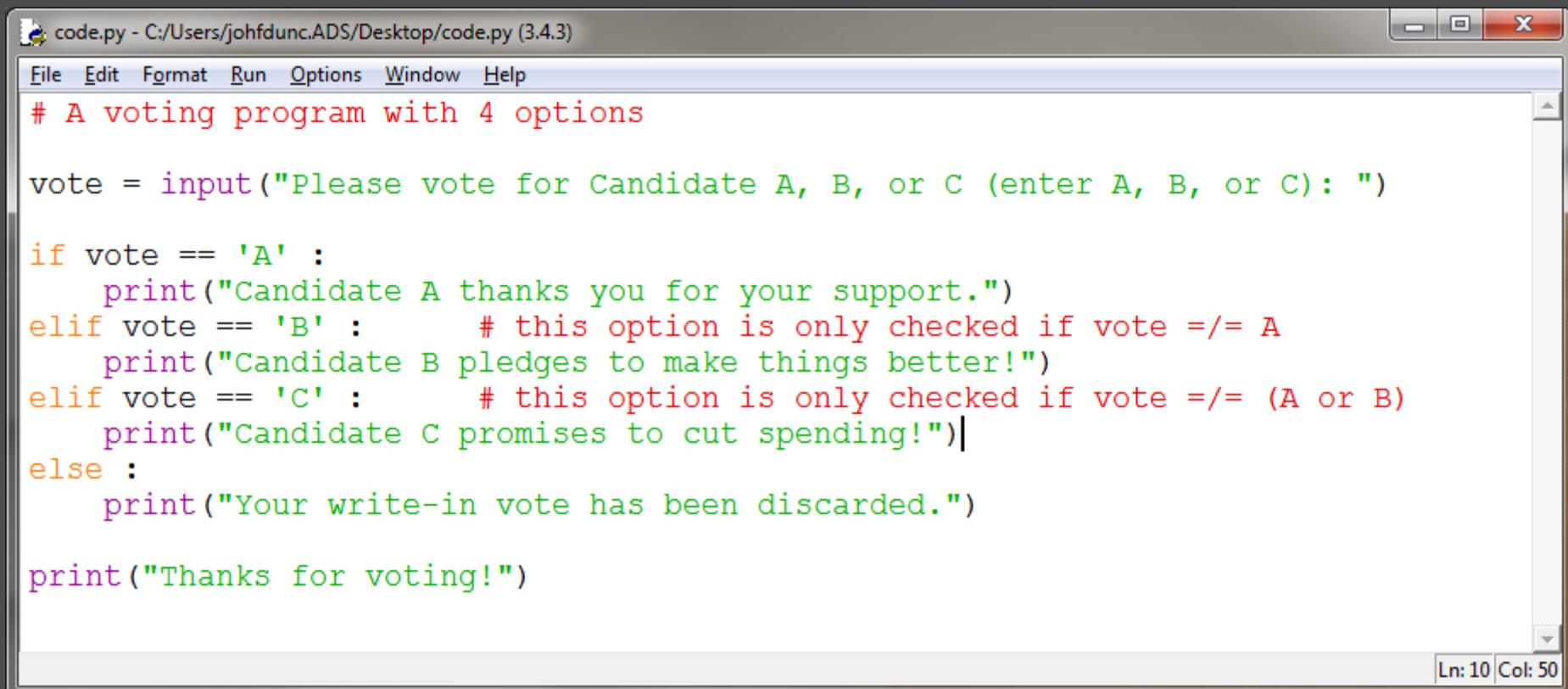
else:

#This will happen only if ALL conditions are false!

#Something that needs to happen all the time

Statements like these are sometimes called “if-then-else” statements in other programming languages)

Voting Program



A screenshot of a Windows-style code editor window titled "code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code in the editor is a Python script for a voting program:

```
# A voting program with 4 options

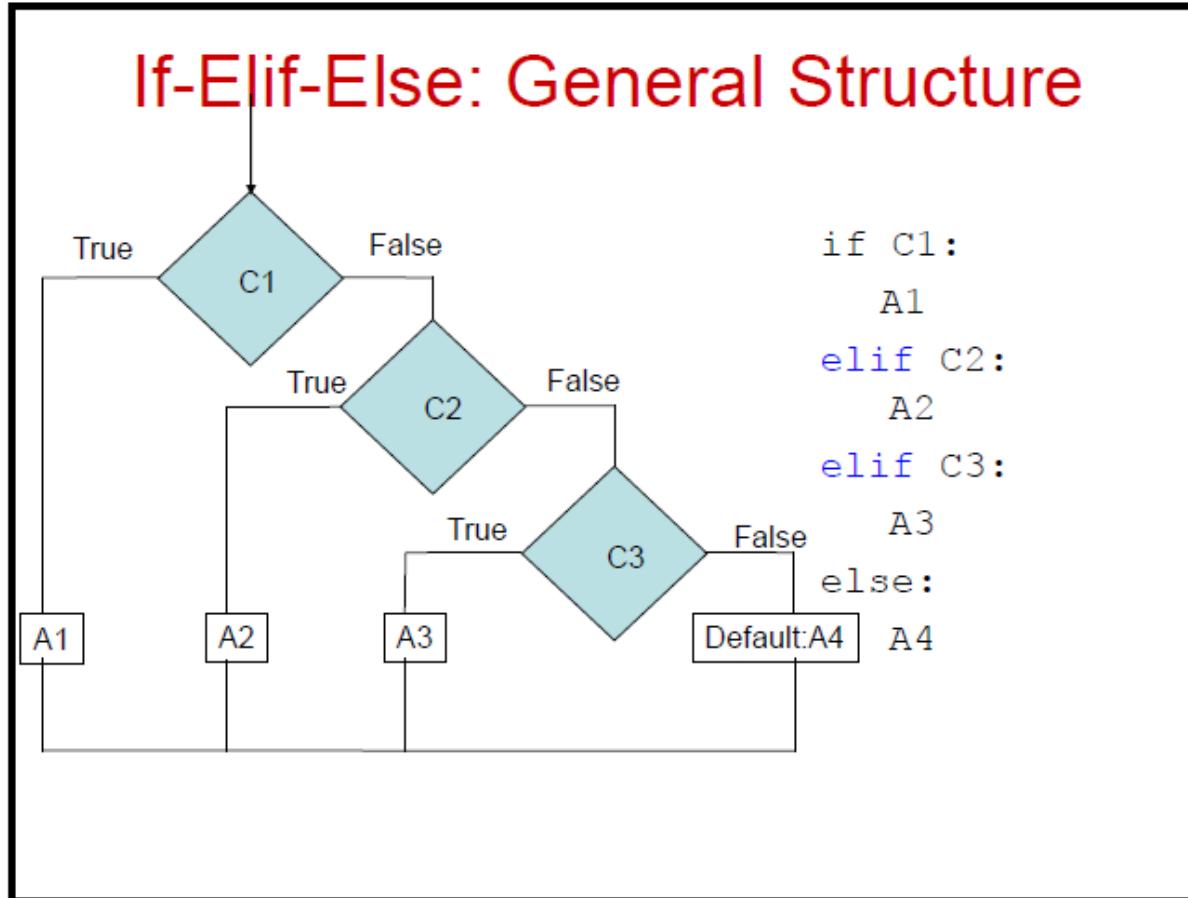
vote = input("Please vote for Candidate A, B, or C (enter A, B, or C): ")

if vote == 'A' :
    print("Candidate A thanks you for your support.")
elif vote == 'B' :      # this option is only checked if vote != A
    print("Candidate B pledges to make things better!")
elif vote == 'C' :      # this option is only checked if vote != (A or B)
    print("Candidate C promises to cut spending!")
else :
    print("Your write-in vote has been discarded.")

print("Thanks for voting!")
```

The status bar at the bottom right shows "Ln: 10 Col: 50".

If / elif / else structures



The first match means we stop!!!

Questions?

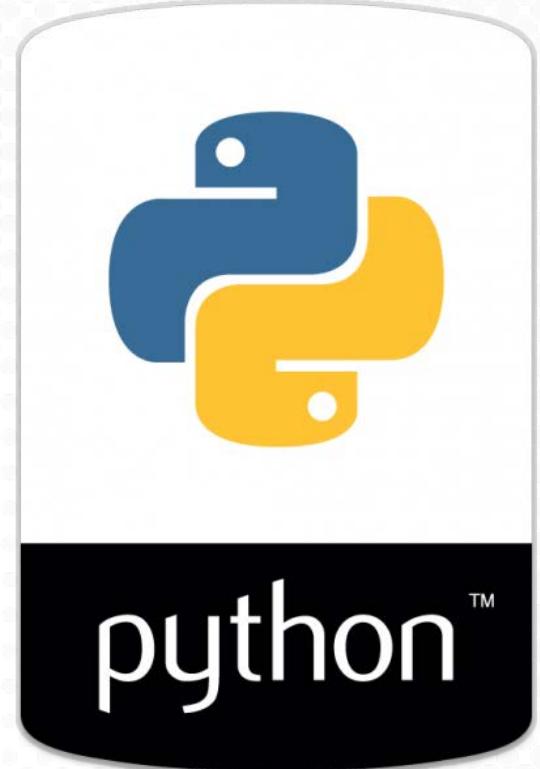
INFORMATICS

Lecture 6

I210 – INTRODUCTION TO
PROGRAMMING WITH PYTHON

Today

- Sequences
- For Loops
- The range() function



Quiz Advice

- Your answers should follow our style advice!
- Surround operators with spaces.

```
result = (number * 3) + 7
```

- Don't put spaces after function names!

```
print(max(numbers) )|
```

If you don't pay attention, you may lose points!

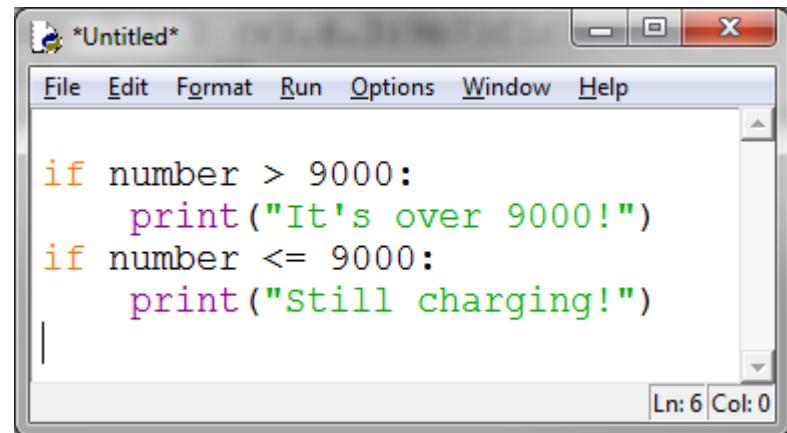
Quiz Advice

- Does the question use checkboxes that allow more than one answer to be selected? *If so, you may need to select more than one answer.*
- Does the questions use radio buttons that allow only one answer to be selected? *Then most likely, only one answer is the BEST answer.*
- Be careful! If we legitimately made a mistake, email Erika at ebigalee@indiana.edu. Say WHAT question and WHAT answer need to be fixed. We'll fix it.

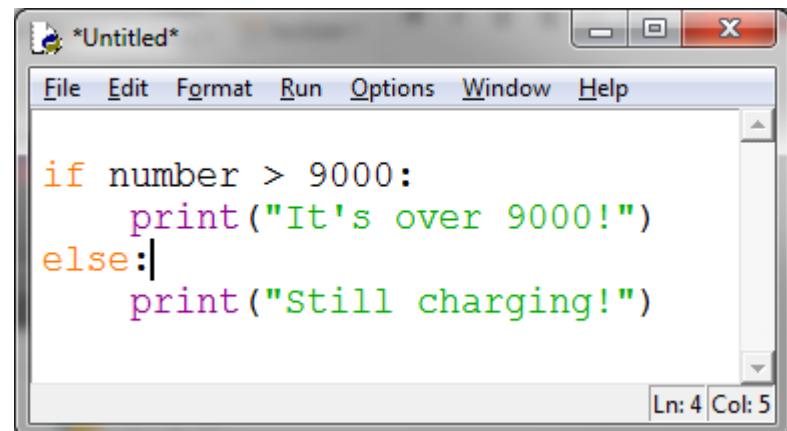
Quiz 2 – Common Mistake

- If you have a situation like this:
- It's better to use **else**:

That way, if the first Boolean condition is True, we don't have to check the second one.



```
if number > 9000:  
    print("It's over 9000!")  
if number <= 9000:  
    print("Still charging!")
```



```
if number > 9000:  
    print("It's over 9000!")  
else:  
    print("Still charging!")
```

Program Planning (Algorithms)

Algorithm: Set of clear, easy-to-follow instructions for solving a problem or accomplishing a task.

1. Start at the highest level – *what does your program need to accomplish?*
2. Then, break those goals into individual steps
3. The steps in your algorithm often become your comments!

We'll show algorithms for our problems before the solutions.

Remember This?

- **Sequence:** An *ordered* list of elements
- **Element:** A single item in a sequence
- **Iterate:** To move through a sequence, *in order*

So *how* do we iterate?

We'll use a structure called a **loop** that can repeat code. First, we'll learn **for loops**.

For loops and sequences

This is the simplest way to use a **for loop** to iterate over a sequence:

for var in sequence:

#code block goes here
#and must be indented

The value of **var** starts out as the first element of the **sequence**, then becomes the next, etc.

Style Note: **var** is a variable you're declaring as part of writing the loop, so follow our rules for variable naming!

For loops and sequences

A list is a *sequence*. So if we had a list like this:

```
names = ['Herb', 'Elizabeth', 'Lars', 'Lisa', 'Owen']
```

We could use a for loop like this:

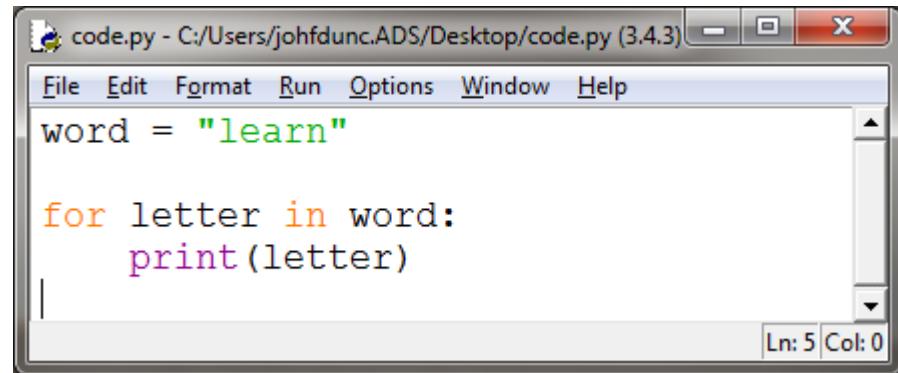
```
for name in names:  
    print(name)
```



```
>>>  
| Herb  
| Elizabeth  
| Lars  
| Lisa  
| Owen  
>>>
```

TIP: Sequence names are often plural. For loops often use a singular form of the word!

For loops, strings, in

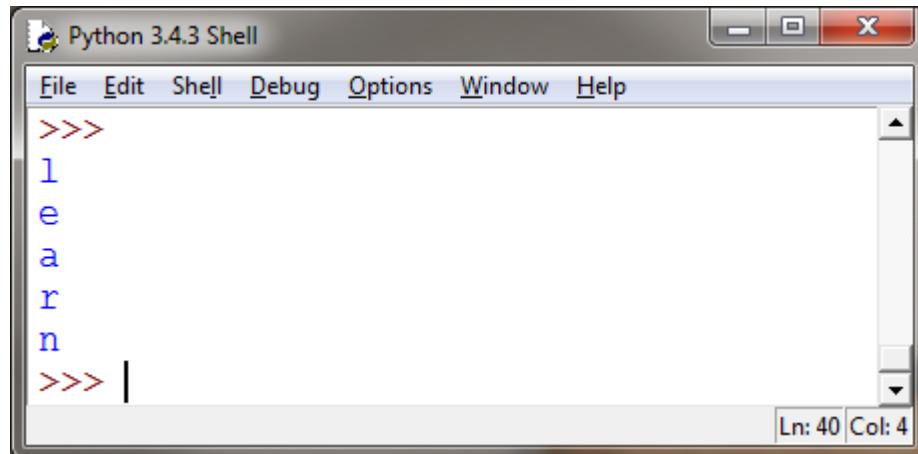


```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
word = "learn"

for letter in word:
    print(letter)
|
Ln: 5 Col: 0
```

A string is also a *sequence*.

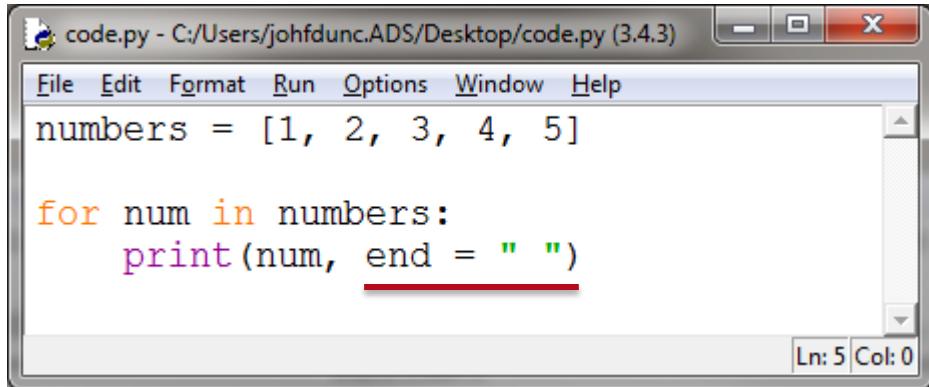
When we loop over it, the *elements* are characters.



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>>
l
e
a
r
n
>>> |
Ln: 40 Col: 4
```

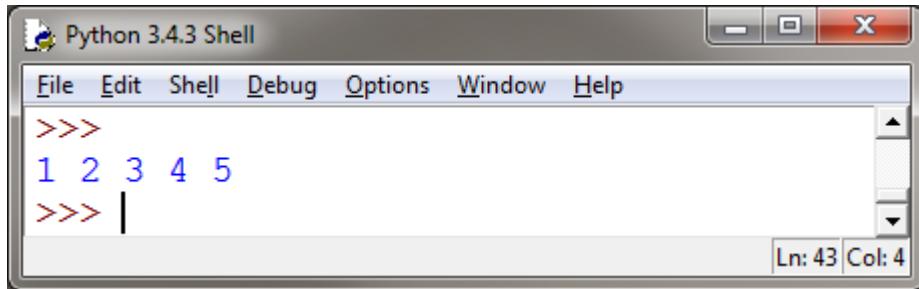
By default, if we print each element, it will be **printed on its own line**.

For loops, strings, in



```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num, end = " ")
```

- Python allows us to manually specify the **end character** in a call to the **print** function.



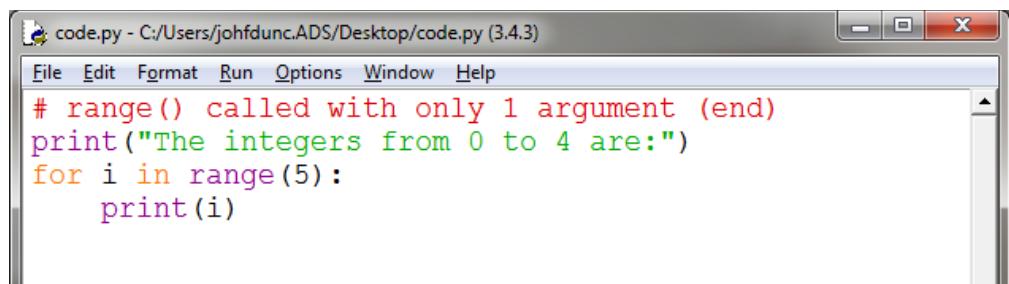
```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>>
1 2 3 4 5
>>> |
```

- By setting it to " " here, we cause all of the output to be printed on the same line.

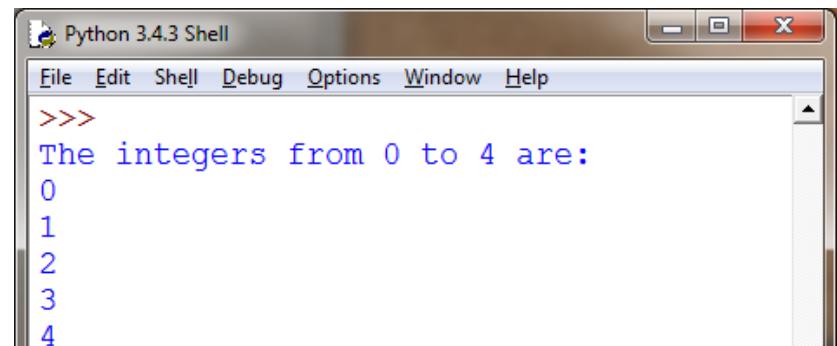
This is called *suppressing the newline*.

The range() function

The **range()** function gives us a **sequence** of integers (a list!). *Here are three ways you can call it:*



```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
# range() called with only 1 argument (end)
print("The integers from 0 to 4 are:")
for i in range(5):
    print(i)
```



```
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>>
The integers from 0 to 4 are:
0
1
2
3
4
```

range() and len()

The **range()** function is often combined with the **len()** function to allow us to loop over the elements of a sequence.

```
# The interaction between the variable
# in a for loop and index numbers.

string = "Python"
for i in range(len(string)):
    print("The letter at index", i, "is", string[i])

print()

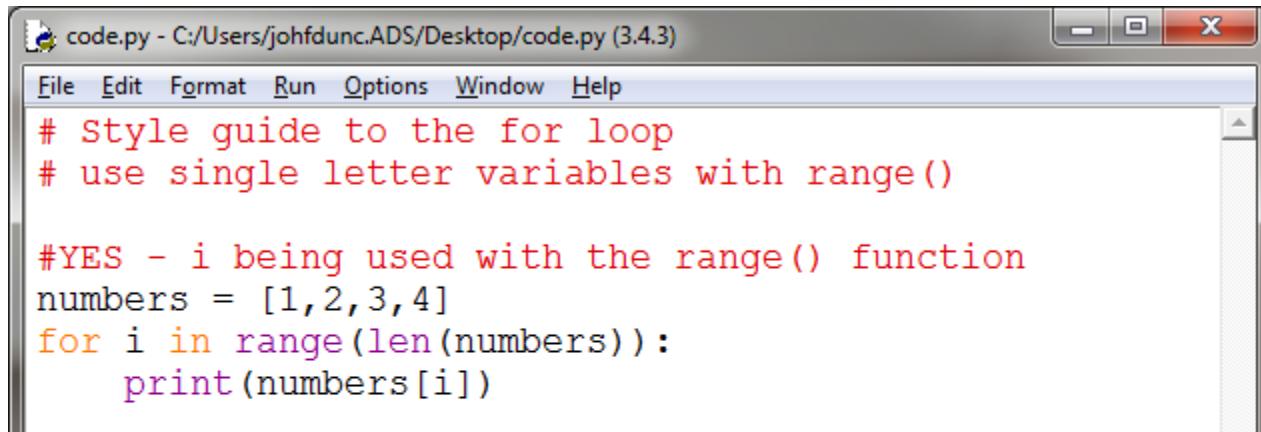
numbers = [1, 2, 3, 4]
for i in range(len(numbers)):
    print("The number at index", i, "is", numbers[i])
```

```
>>>
The letter at index 0 is P
The letter at index 1 is y
The letter at index 2 is t
The letter at index 3 is h
The letter at index 4 is o
The letter at index 5 is n
>>>
The number at index 0 is 1
The number at index 1 is 2
The number at index 2 is 3
The number at index 3 is 4
>>>
```

TIP: `print()` by itself produces a blank line.

Single Letter Variables

Variables with single letter names (like **i**, **j**, or **k**) are, by tradition, *only used when they represent index numbers in a loop.*



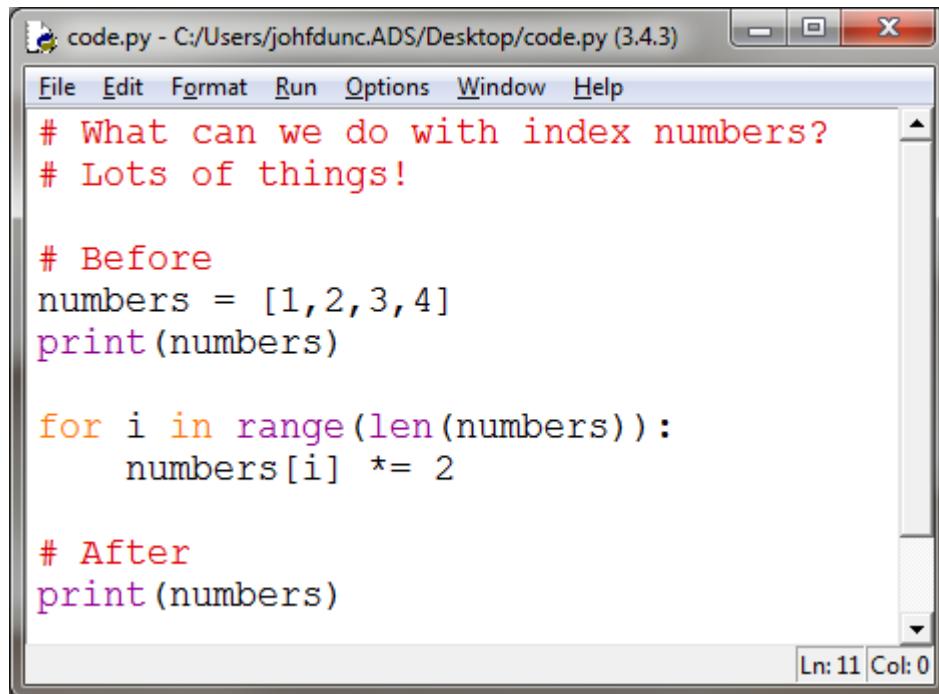
```
code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)
File Edit Format Run Options Window Help
# Style guide to the for loop
# use single letter variables with range()

#YES - i being used with the range() function
numbers = [1,2,3,4]
for i in range(len(numbers)):
    print(numbers[i])
```

This is confusing because an

Index Numbers and For Loops

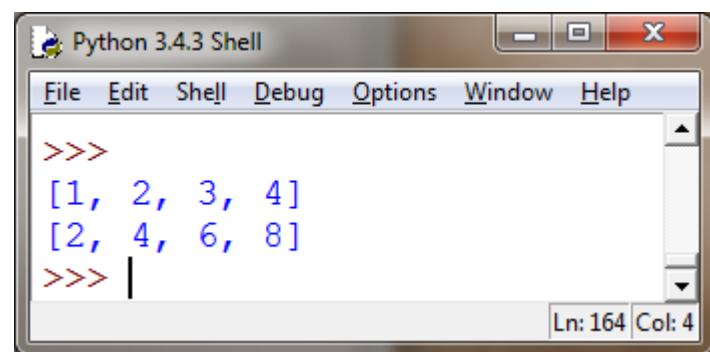
Combining index numbers and for loops allows us to change things in a list with a loop!



code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)

```
# What can we do with index numbers?  
# Lots of things!  
  
# Before  
numbers = [1, 2, 3, 4]  
print(numbers)  
  
for i in range(len(numbers)):  
    numbers[i] *= 2  
  
# After  
print(numbers)
```

Ln: 11 Col: 0



Python 3.4.3 Shell

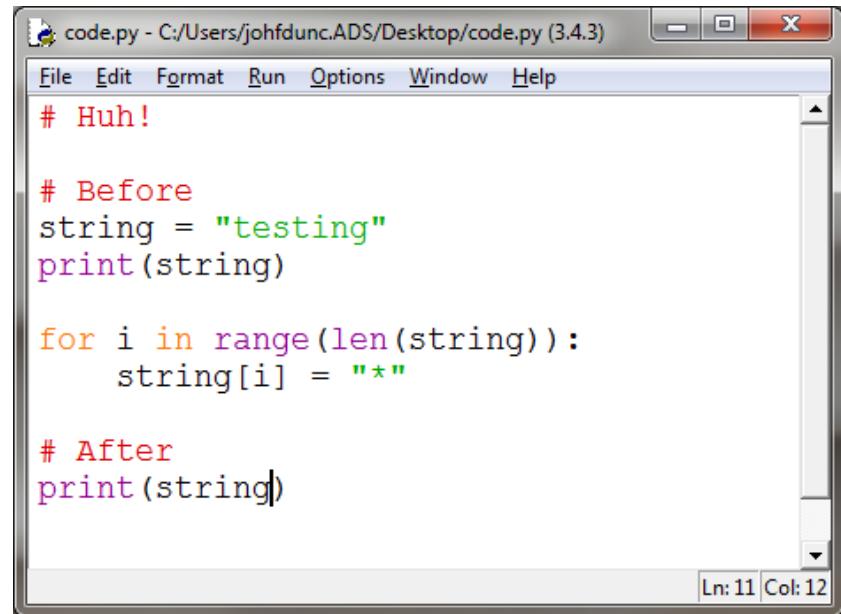
```
>>>  
[1, 2, 3, 4]  
[2, 4, 6, 8]  
>>> |
```

Ln: 164 Col: 4

Index Numbers and For Loops

We can't change strings
the same way, however:

Remember, *strings and tuples are immutable!*



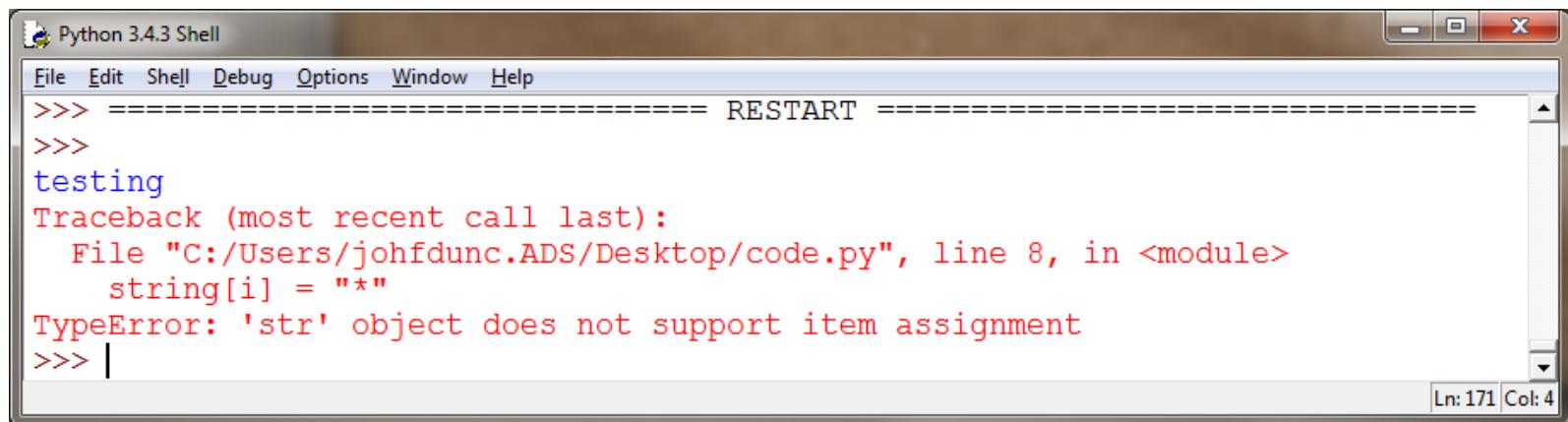
A screenshot of a Windows-style code editor window titled "code.py - C:/Users/johfdunc.ADS/Desktop/code.py (3.4.3)". The menu bar includes File, Edit, Format, Run, Options, Window, and Help. The code in the editor is as follows:

```
# Huh!
# Before
string = "testing"
print(string)

for i in range(len(string)):
    string[i] = "*"

# After
print(string)
```

The status bar at the bottom right shows "Ln: 11 Col: 12".



A screenshot of the Python 3.4.3 Shell window. The title bar says "Python 3.4.3 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The shell prompt shows the user entering code and receiving an error message:

```
>>> ===== RESTART =====
>>>
testing
Traceback (most recent call last):
  File "C:/Users/johfdunc.ADS/Desktop/code.py", line 8, in <module>
    string[i] = "*"
TypeError: 'str' object does not support item assignment
>>> |
```

The status bar at the bottom right shows "Ln: 171 Col: 4".

Index Numbers and For Loops

There are other ways to use the length of a string to make a new string!

```
# 'Censoring' a string

# Before
string = "testing"
print(string)

string = "*" * len(string)

# After
print(string)
```

```
>>>
testing
*****
>>>
```

The 2 types of for loop

1 `for char in word:`

```
#do something to each letter  
#where we don't care about  
#the index position
```

Use a for loop over a sequence if you just need to do something to each element, and it doesn't require you to know where you are in the sequence.

2 `for i in range(len(sequence)):`

```
#use an index number  
#to accomplish something  
#single letter variable names
```

Use a for loop with range and a numeric loop control variable if you need to know where you are in a sequence.

Single letter variable names here only!

Questions?

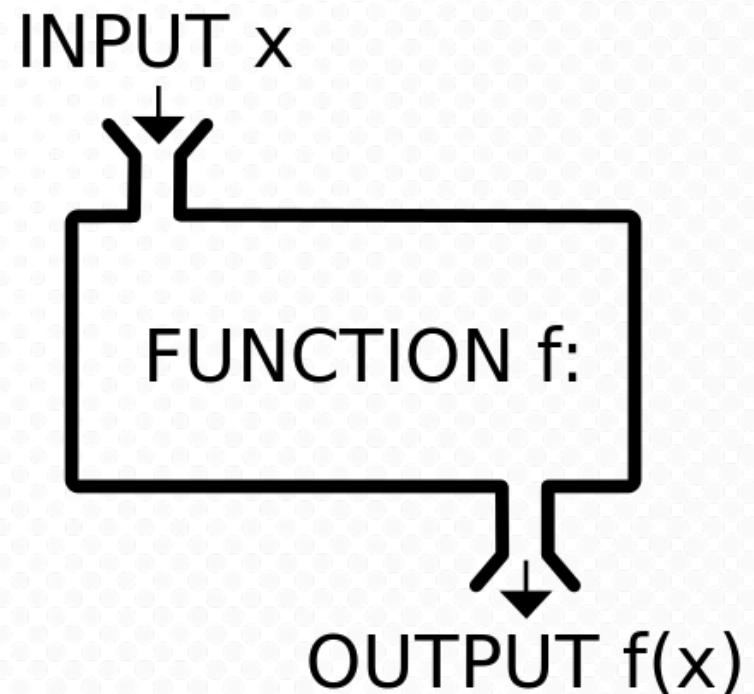
INFORMATICS

Lecture 7

I210 – INTRODUCTION TO
PROGRAMMING WITH PYTHON

Today

- Functions
- The return statement
- Multiple return values



Functions

So far we've seen many built-in functions that let us do complicated operations easily:

print()

int()

input()

str()

len()

float()

range()

eval()

We can also create our own functions!

Why write your own functions?

Divide and conquer

- Break a complicated problem into simpler problems
- Encapsulated Testing – test only one piece at a time!

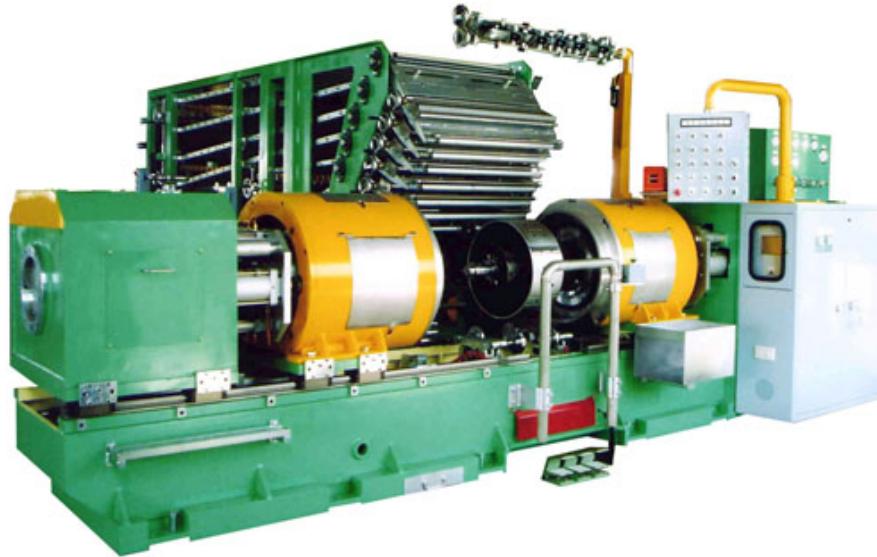
Avoid writing redundant code

- Copying and pasting code is almost always a bad idea!
- Write a re-usable tool: create a function for the job.

Legibility

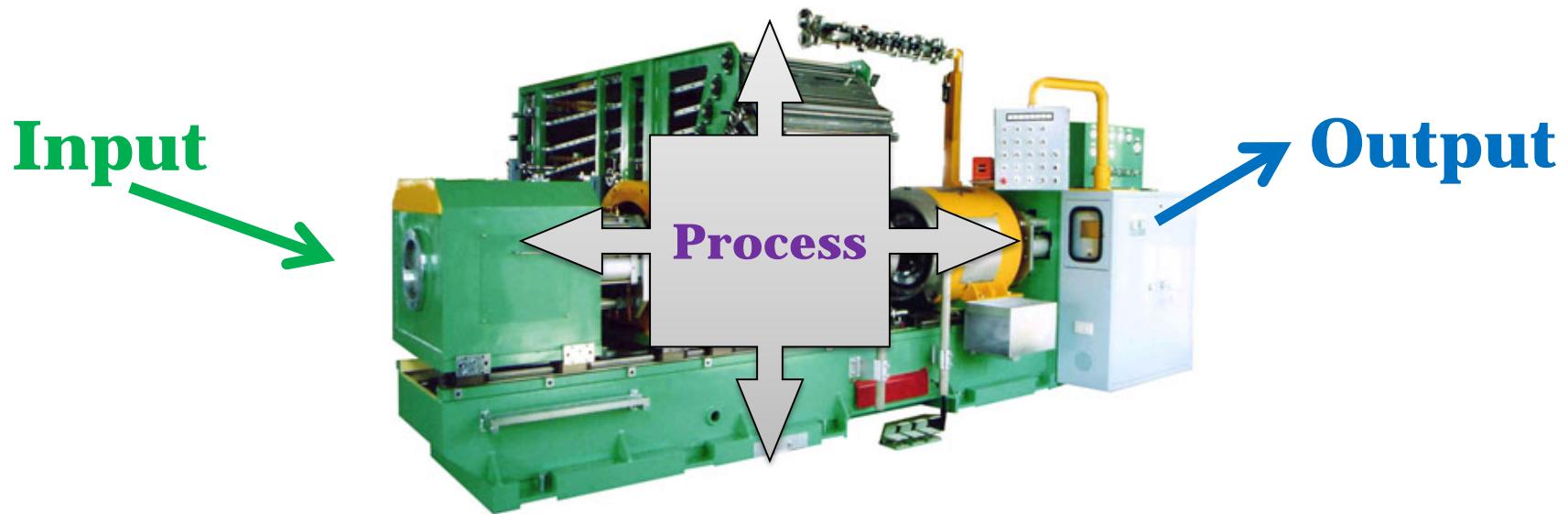
- 90 lines of code or 3 functions of 20 - 40 lines?

A function is like a machine...



- A machine is designed to do a specific task
- Machines take time to build, but save it in long run
- You can use a machine without having to understand exactly how it works inside
 - You just have to know how to *interact* with it!

A function is like a machine...



Machines take *input* and produce *output*

- A user only has to know what kind of input it needs, and what kind of output it produces
- The machine's creator has to also design the *process* that goes on inside the machine

Function Metaphor

Let's say our machine
engraves key chains.

- The **input** will be a word or phrase.
- The **process** describes how to take each piece of the input and engrave it onto the keychain.
- The **output** will be the engraved keychain.

We don't want to output the first half of the keychain, then the second half.



Function Components

Likewise, functions are defined by 3 components:

- 1. Arguments (input)**

Information the function needs to be able to do its job.

- 2. Function Body (process)**

The code inside the function that does the job.

- 3. Return Values (output)**

Information the function sends back.

Functions

To create a new function, we begin by writing a ***definition***.

```
def function_name( argument1, argument2, etc... ):  
    #code for function goes here  
    #more code, etc  
    return return_value    #sends this back!
```

Function parts:

Arguments, Function Body, Return Values

Arguments are like variables!

Functions

- All of your function definitions should go at the *top of your program, under imports.*
- The section of your code after that is called the **main section** or **main body** of the code (*commonly abbreviated “main”*).

Outline for a Python Program

#import statements, if any

#function definitions, if any

#main section of the program

#calls to our functions

#may occur here

#function definitions do *not* occur here!!

Functions

Not all functions take **arguments**.

```
def function_name():
    #code for function goes here
    #more code, etc
    return return_value    #sends this back!
```

Functions

Not all functions **return values.**

```
def function_name(argument1, argument 2, etc.. ):  
    #code for function goes here  
    #more code, etc  
print or write to a file
```

Writing a Function

Bigger problem:

How would I write a function to tell if the difference between two numbers is odd?

Simpler:

How would I write a function to tell me if a number is odd?

Functions

#function definitions

```
def odd(number):
```

To get started, I decide on a name for my function. It should describe what the function does. The definition will go above the main section.

I also decide:

1. How many pieces of information (arguments) do I need?
2. What kind of information? I name accordingly.

Functions

```
#function definitions  
def odd(number):  
    if number % 2 == 1:  
        result = True  
    else:  
        result = False  
    return result
```

```
#main section  
print(odd(6))
```

```
if odd(5):  
    print("5 is odd")
```

number is the function's argument. When we call the function, we have to pass it an actual value for number!

result is the function's return value. When we call the function, it will return either **False** or **True**.

So **odd** is a machine that takes a number as input, and outputs True or False.

Function example

```
#function definitions  
def odd(number):  
    if number % 2 == 1:  
        result = True  
    else:  
        result = False  
    return result
```

```
#main section  
print(odd(6))
```

```
if odd(5):  
    print("5 is odd")
```

The main section does not know how **odd()** works – it just knows it needs a number as input and returns True or False.

```
#function definitions  
def odd(number):  
    return number % 2 == 1
```

```
#main section  
print(odd(6))
```

```
if odd(5):  
    print("5 is odd")
```

Later, we could make the code in **odd()** more efficient, and the main section stays the same!

Functions

```
#function definitions
```

```
def odd(number):
```

```
    return number % 2 == 1
```

odd_diff()

has two arguments!

```
def odd_diff(num1, num2):
```

```
    difference = num1 - num2
```

```
    return odd(difference)
```

If one of our functions calls another, it needs to appear *below it* in our code!

```
#main
```

```
user_num1 = eval(input("Please enter an int: "))
```

```
user_num2 = eval(input("Please enter an int: "))
```

```
print(odd_diff(user_num1, user_num2))
```

Common confusion: print vs return

print sends a value to the user

return sends a value to another part of the program

The other part of the program could print the value, or might do something else with it (write it to a file, use it as input)

These two programs do exactly the same thing. Is one better?

```
# program 1
def square(number):
    return number * number
```

```
number = 3
print(square(number))
```

```
# program 2
def square(number):
    print(number * number)
```

```
number = 3
square(number)
```

This function is
more general! We
can use it whether
or not we want to
print out the result.

Return

The **return statement** always ends the function's execution; code after a return is never run.

You can have *more than one return statement*, however:

```
if (X):
    return a
    #don't put code here! It will never run!
else:
    #code to run if X was False
    #more code
    return b
```

- If there's *no return statement*, the function returns **None**.

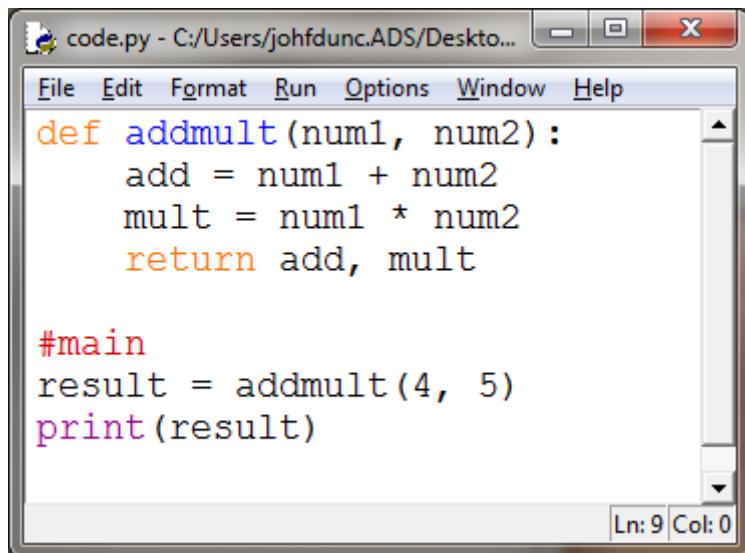
Return Values

You don't have to *catch* the return value from your function (assign it to a variable), but you often want to.

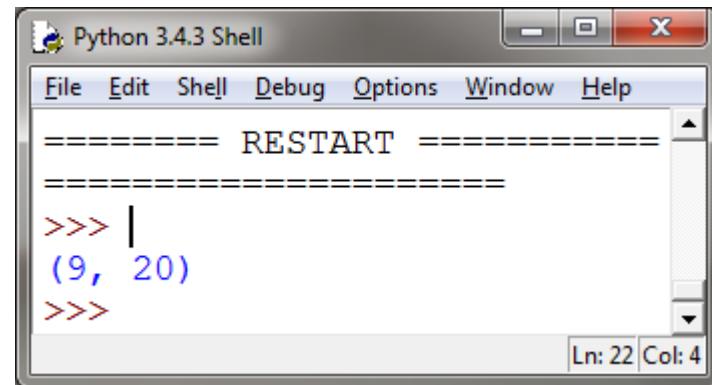
- You can use it just like you would any other value in Python.
 - E.g. print it, use it in a Boolean condition, etc.

Multiple Return Values

Python allows **multiple return values**:



```
code.py - C:/Users/johfdunc.ADS/Desktop/Pytho...  
File Edit Format Run Options Window Help  
def addmult(num1, num2):  
    add = num1 + num2  
    mult = num1 * num2  
    return add, mult  
  
#main  
result = addmult(4, 5)  
print(result)  
  
Ln: 9 Col: 0
```



```
Python 3.4.3 Shell  
File Edit Shell Debug Options Window Help  
===== RESTART =====  
=====  
=>>> |  
(9, 20)  
>>>  
Ln: 22 Col: 4
```

- They are sent back in a tuple!

Encapsulation / Local variables

Variables created inside a function are called **local variables**. They are only accessible inside the function! You must return them if you want to use them elsewhere.

```
def addmult(num1, num2):  
    add = num1 + num2  
    mult = num1 * num2  
    return add, mult
```

#main section

```
result = addmult(4, 5)  
print(add)
```

This is a *feature*, not a limitation!
Remember: the code calling a function
doesn't need to know how the function
works inside, so it shouldn't have access
to its internal (local) variables.

>>>

```
Traceback (most recent call last):  
  File "C:/Users/J/Desktop/code.py", line 8, in <module>  
    print(add)  
NameError: name 'add' is not defined
```

Questions?