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")

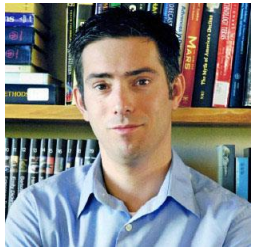
- PhD, Indiana University
- Areas: Education, Security, Privacy, Social Informatics, Pervasive Computing, AI, Cognitive Science

- 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

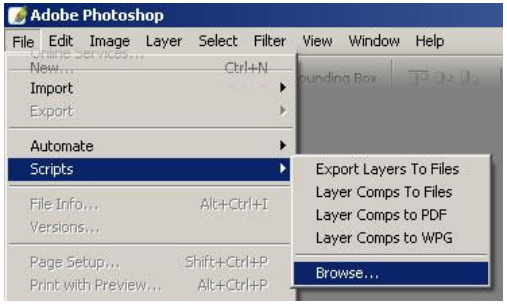
Making them do whatever we want, even if it's something no one has ever done before

Learning to program will help you:

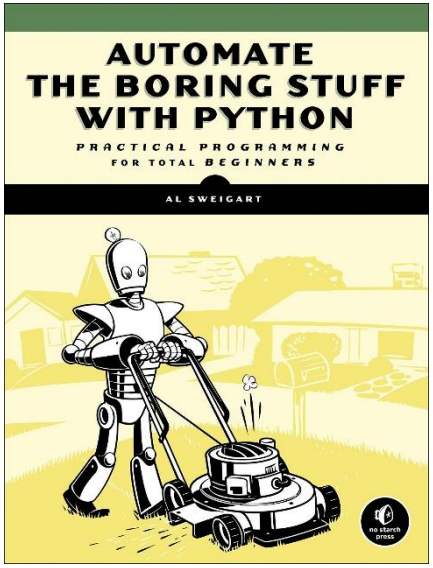
- Create webpages, software, apps, games, etc.
- Analyze research data, optimize business strategies, etc.
- Understand how computers work, and their capabilities and limitations
- Be successful! Have fun! Get a job!

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!

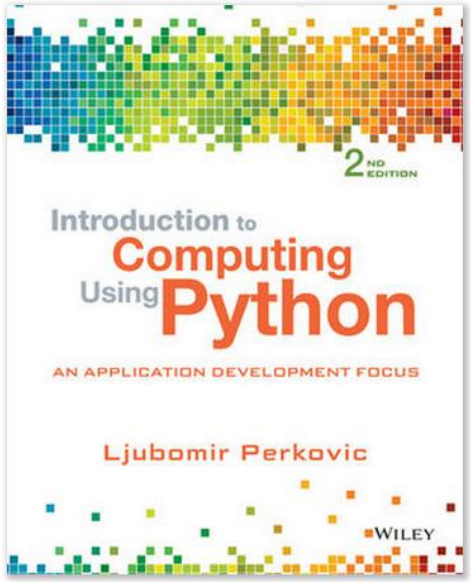
Peer Led Team Learning

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%

- Lecture Work (20%)
  - In teams, 1 submission – show it to us.
- Lab Tasks (15%)
  - In teams, 1 submission – show it to us.

65%

- Retrospective Quizzes (20%)
  - Done out of class on Canvas, open course materials.
- Individual Homework (20%)
  - Drop lowest 1, done *individually*!
- Individual Lab Practicals (35%)
  - 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***

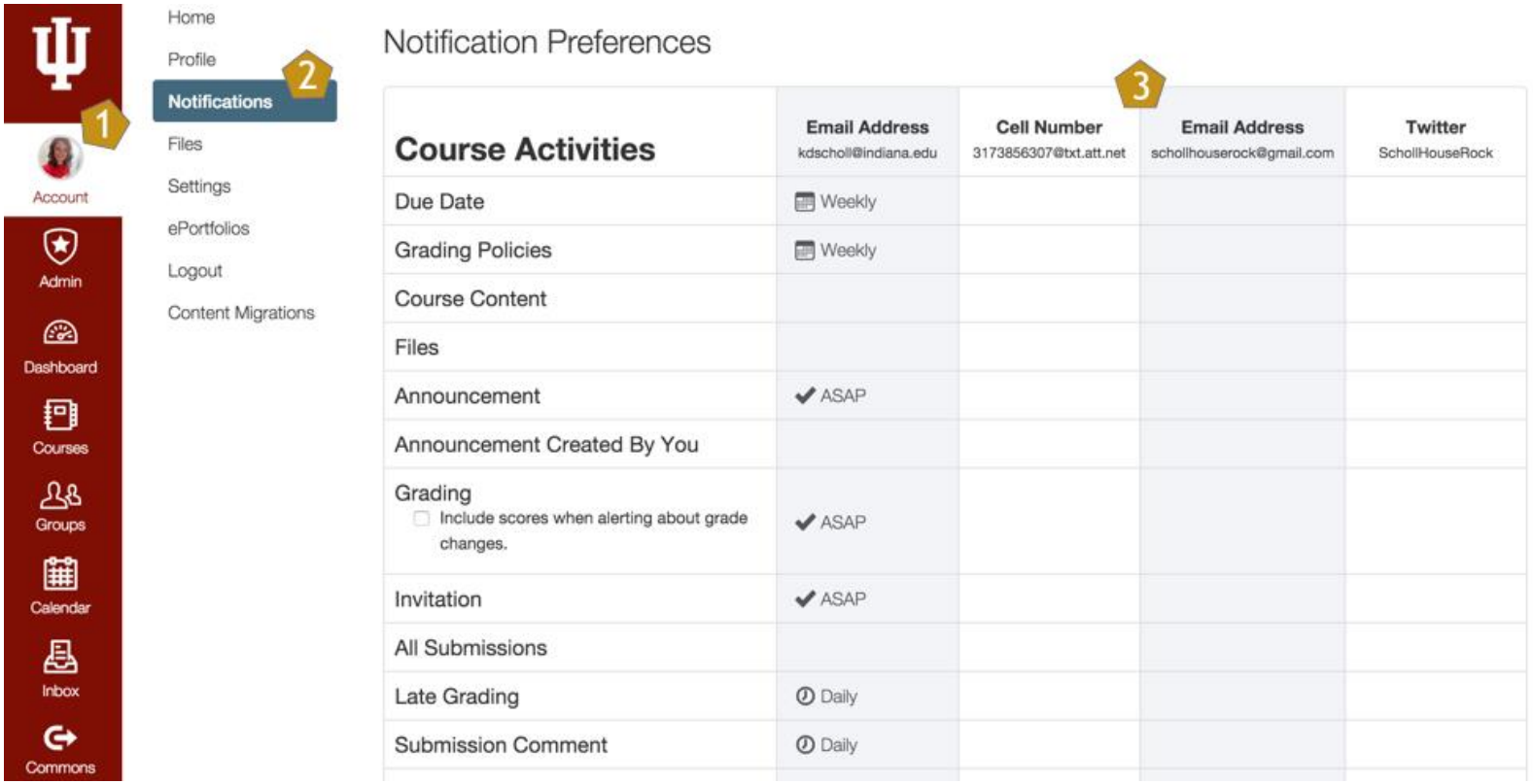
- Actively participate in class, labs, assignments
- Get help when you don't understand

Take advantage of all course resources!

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

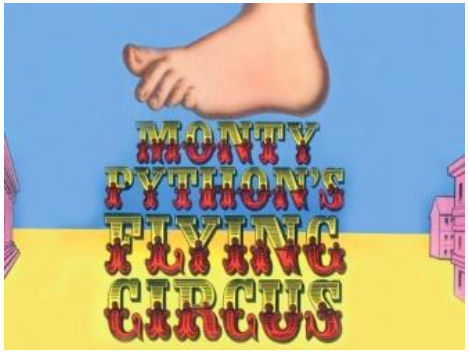
Announcements are Crucial!

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



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()
```

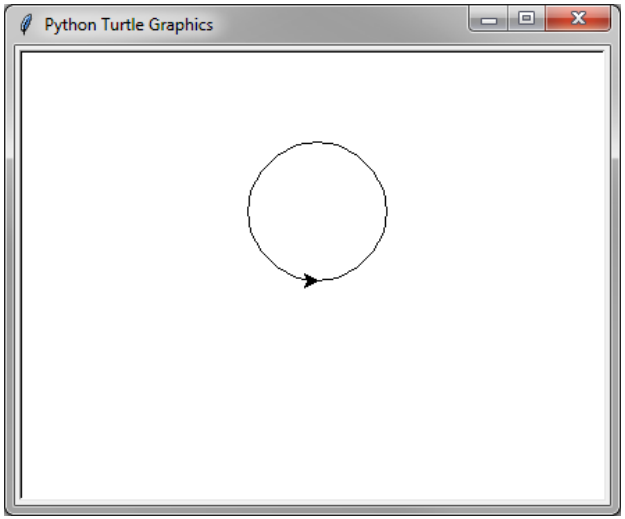
- Java Program

```
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.

This is why we covered algorithms in I201!

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.

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 **D**eve**L**opment **E**nvironment (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

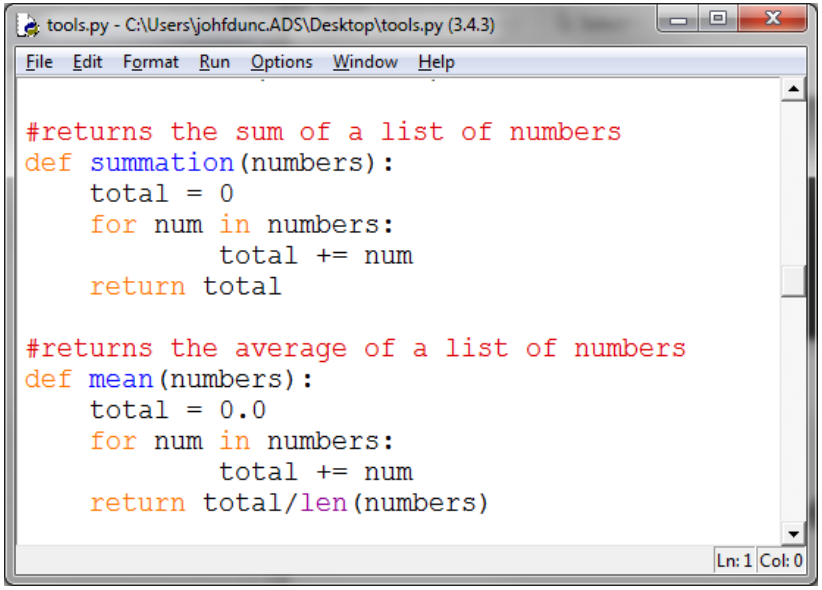
*Python cares about **whitespace**!*

- New lines (hitting Return/Enter) and tabs

Jargon

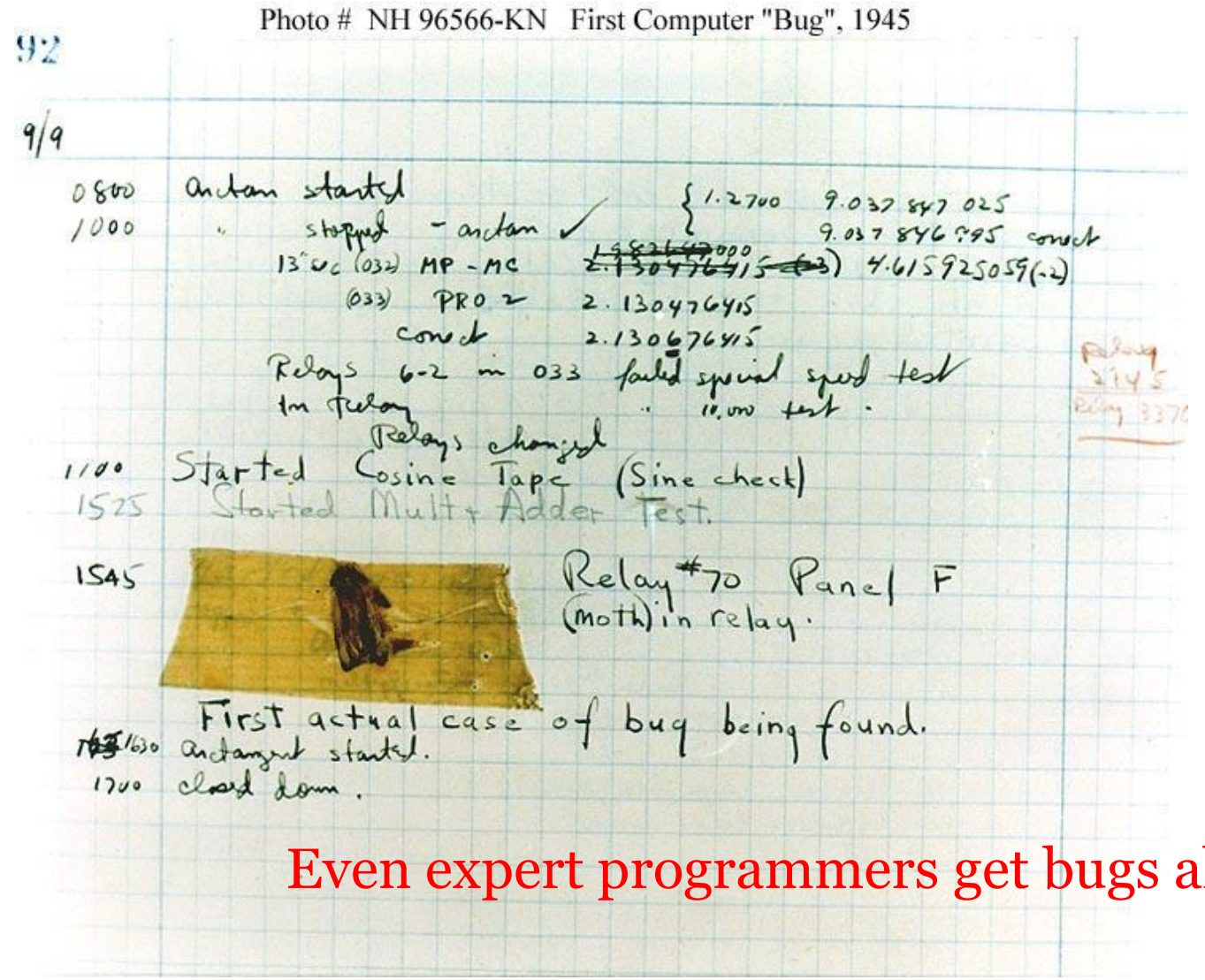
Syntax highlighting:

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



Bug: Error in programming code

Bugs! You will get lots of them!



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?

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

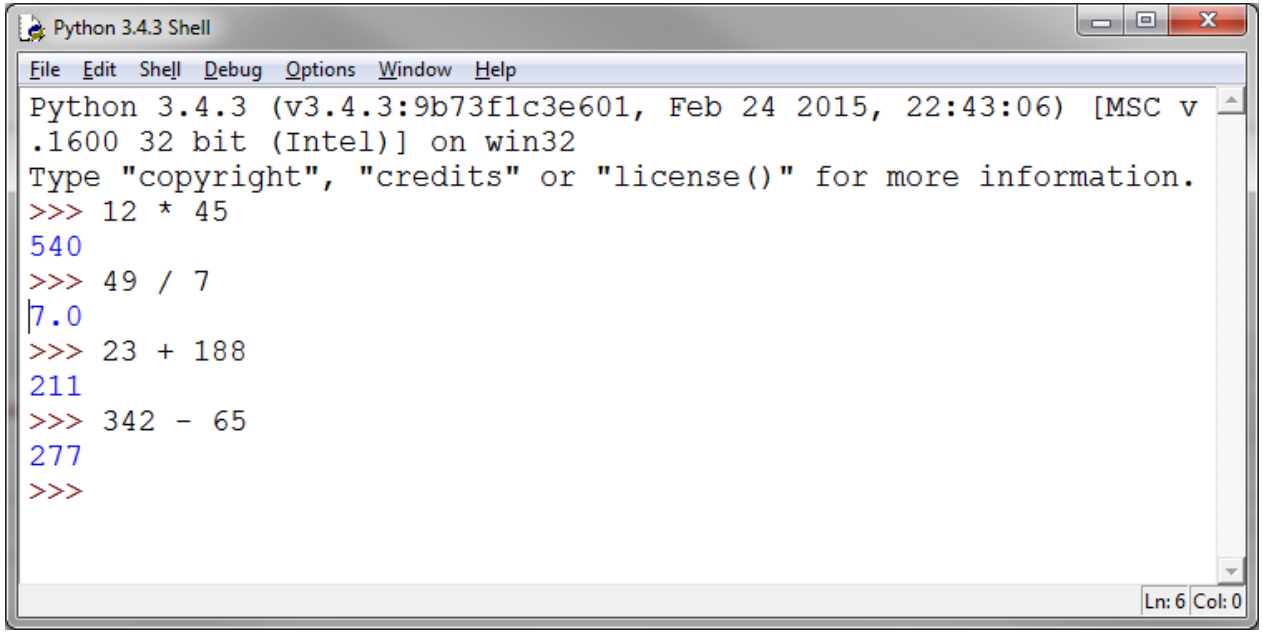
- Posted to Canvas
- If your lecture is using assigned seating for groups, this information is also posted.
- Make sure your group gets checked off when you're done with a problem! (At that point, you're allowed to use your devices to check email, etc).

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!):



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
+	Addition	7 + 3	10
-	Subtraction	7 – 3	4
*	Multiplication	7 * 3	21
**	Exponentiation	7 ** 3	343
/	Division	7 / 3	2.33333333…
//	Integer by Integer Division (rounds down)	7 // 3	2
%	Modulus (remainder under integer division)	7 % 3	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**.

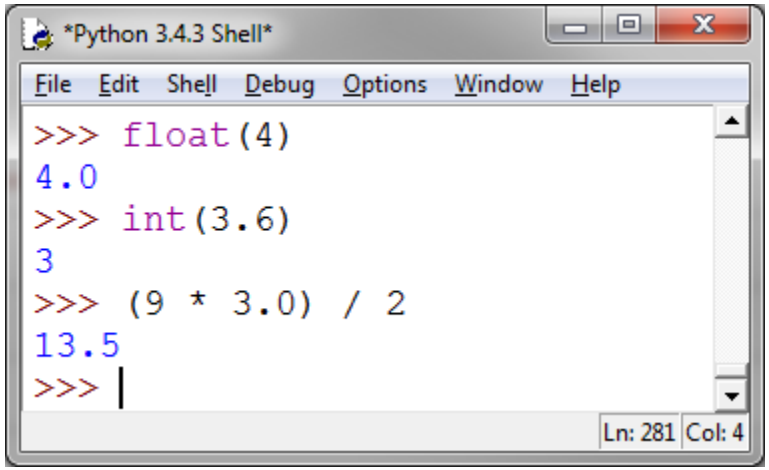
```
>>> abs(-10)
10
```

- **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.

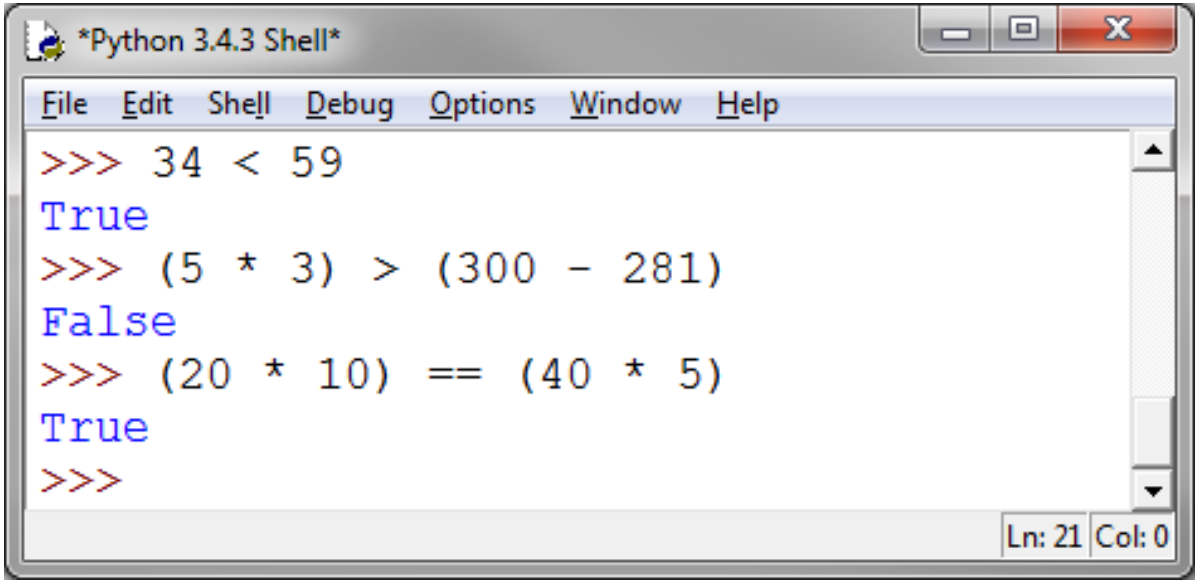


Boolean Values

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:



Boolean Comparison Operators

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

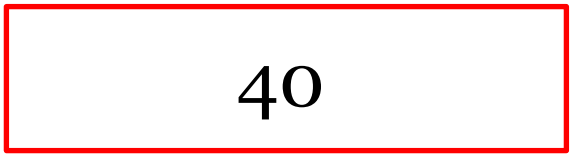
Example

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



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:

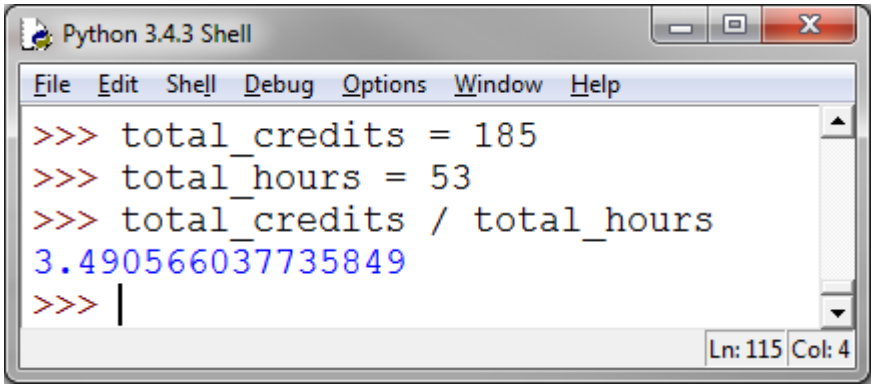
- Is descriptive! (**name** vs **n**)
- Is consistent! (**newPrice** or **new_price**, pick one)
- Observes conventions (start with lowercase letter)
- Isn't too long (try for no more than 15 characters)

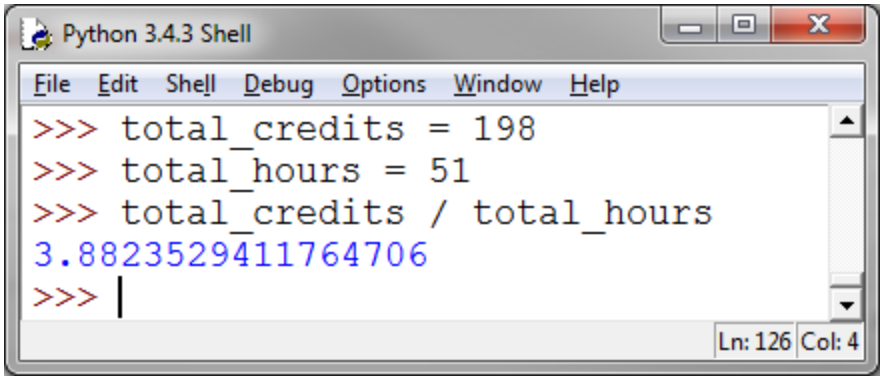
- Remember, case matters!!!
  - **my_name** and **my_nAme** ARE NOT the same!!

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:





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:

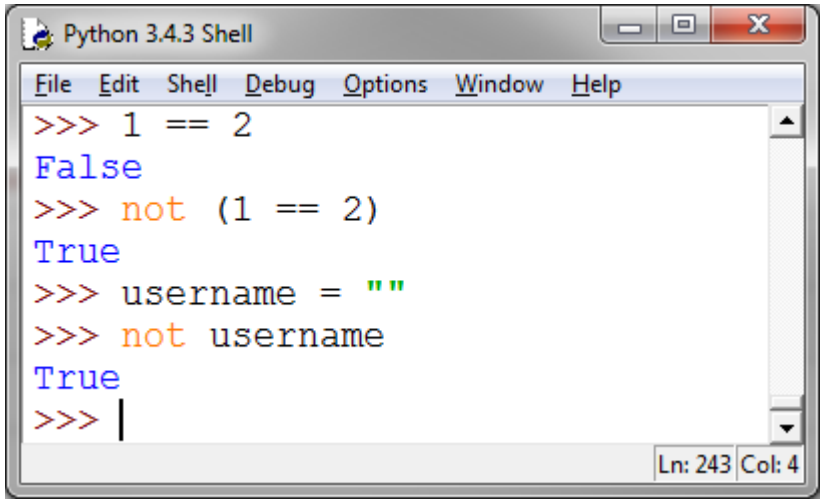
- Display an error message if a user does NOT enter their username
- Allow user to log in if username AND password are correct
- Notify staff if a break-in is detected OR if an incorrect password is entered 3 times

NOT

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

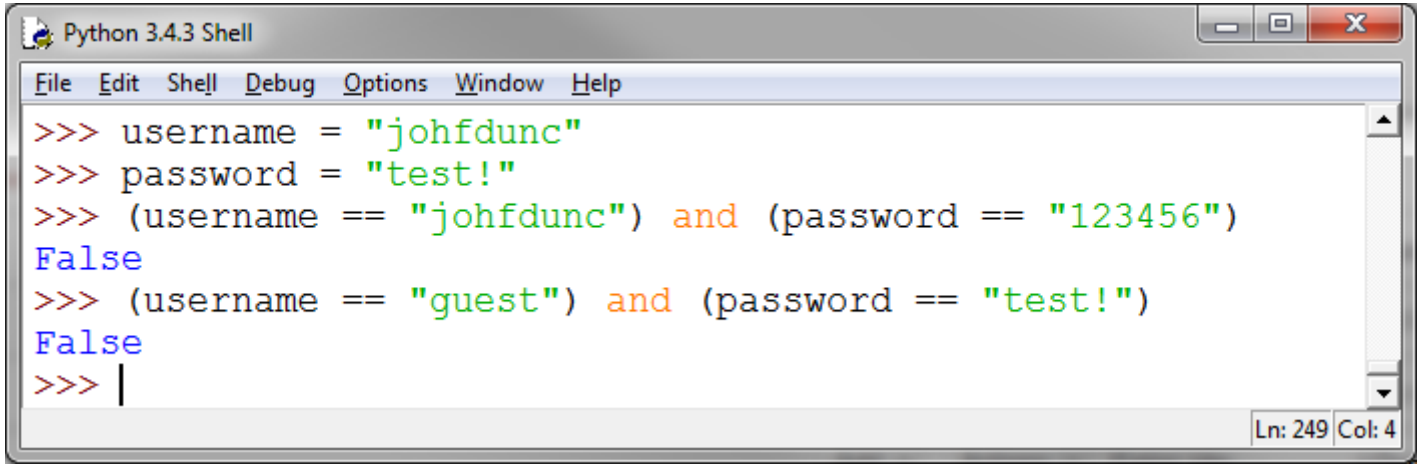
TAKE NOTE:

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



AND

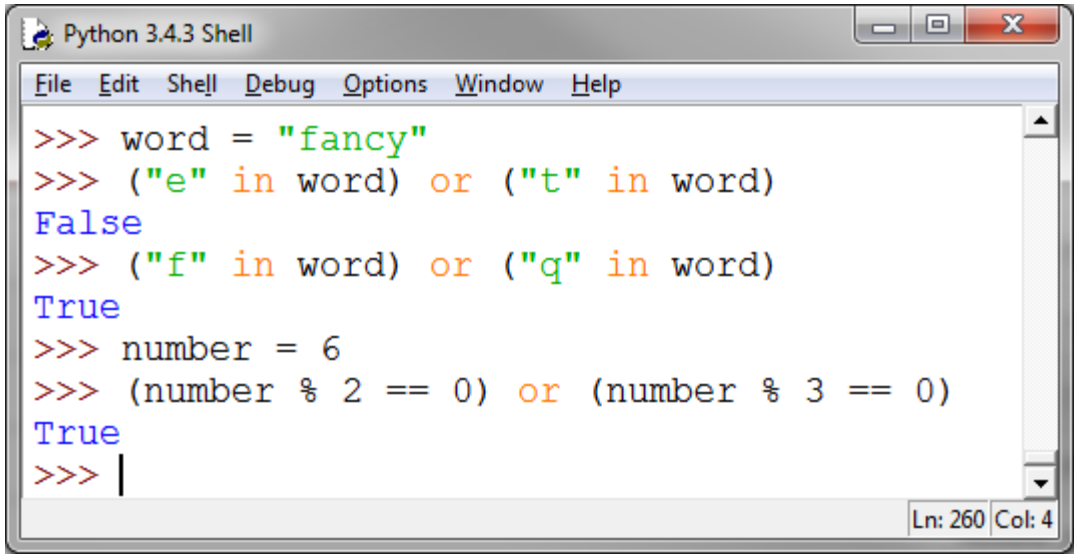
and is a logical operator that connects two Boolean Conditions. It's **True** if they are BOTH true, and **False** otherwise.



- 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!



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 ∨ q	p ⊕ 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?

Lecture 3

I210 – INTRODUCTION TO PROGRAMMING WITH PYTHON

Today

Announcement: PLTL Office Hours

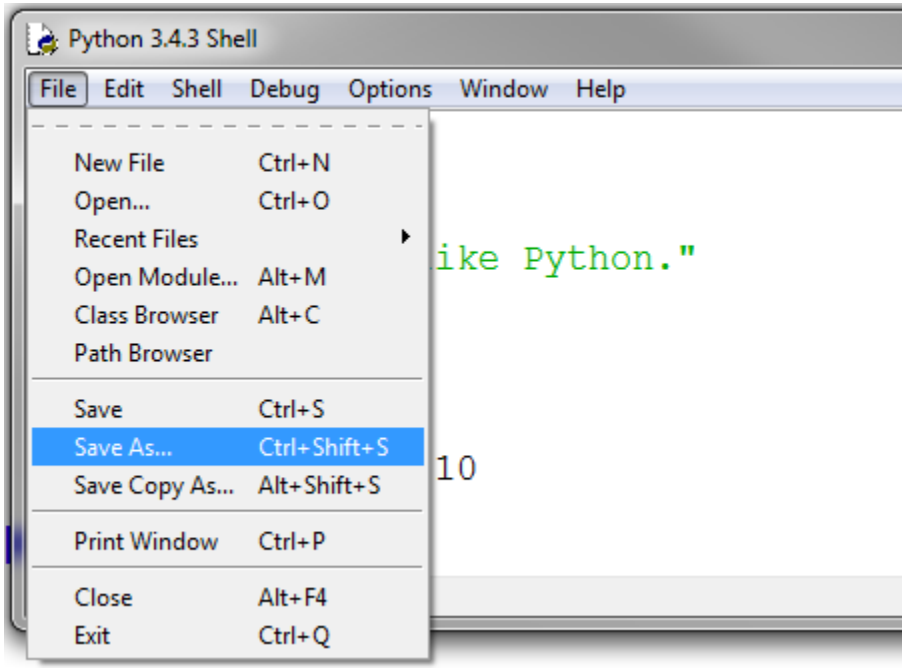
- **strings** (and concatenation) – *New data type*
- **in operator**
- **len function**
- **indexing strings**
- **lists** (changing, deleting, min/max/sum, appending, removing, sorting)
- **methods *versus* functions**

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:

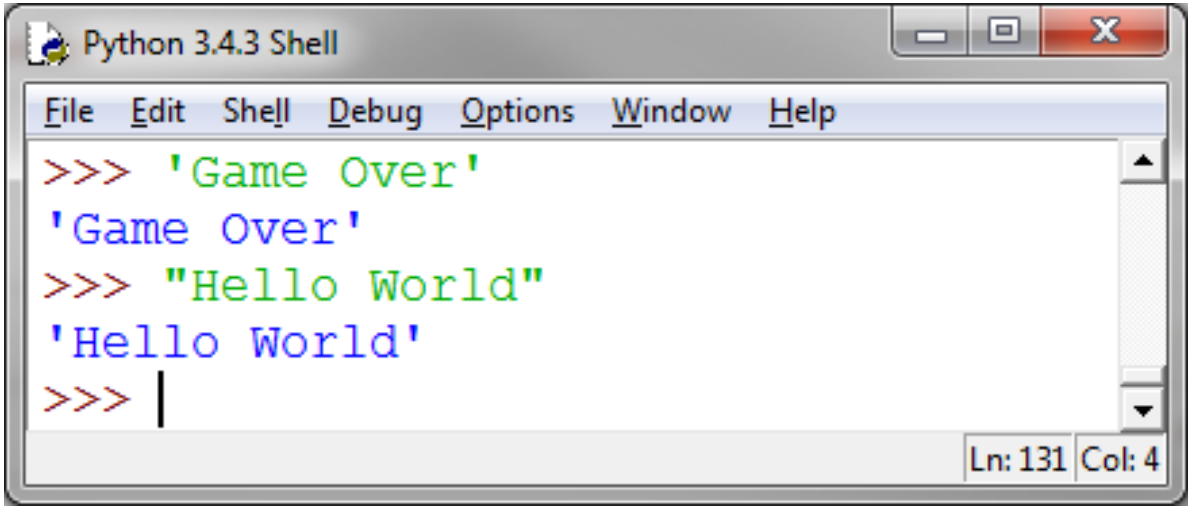


- Save it as a text file (.txt file extension)
- This saves a record of the entire session. It won't allow you to re-run the code, however.

Strings: Using Quotes

Strings (text values) in Python use quotes.

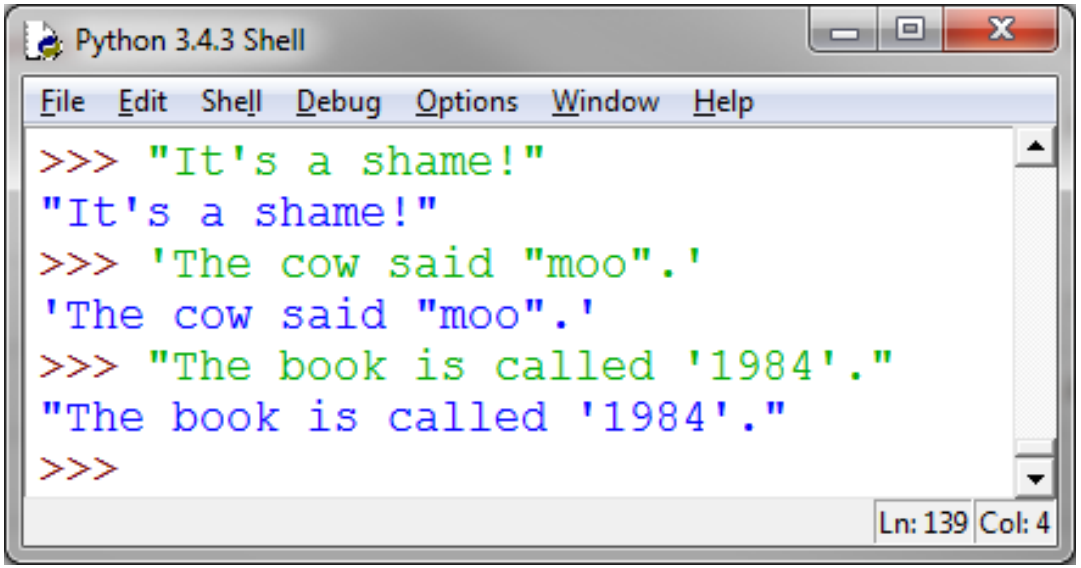
- Either single quotes ' or double quotes " work!
- Start *and* end with the same quotes:



Strings: Using Quotes

Why choose one quote over the other?

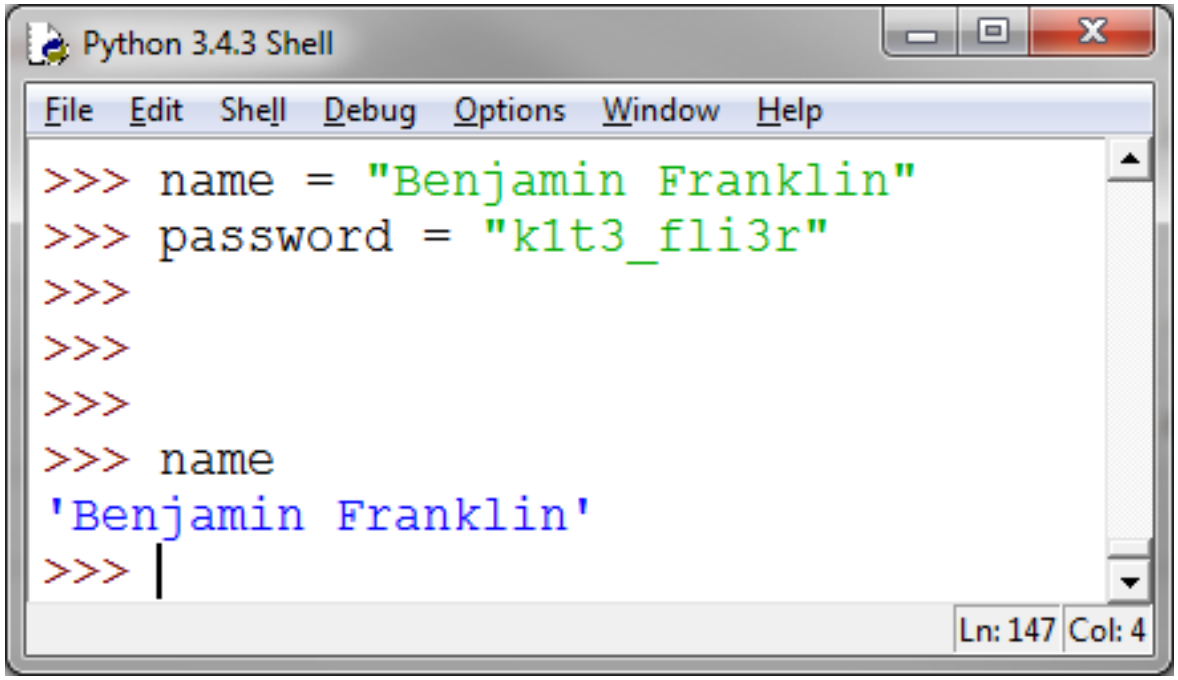
- If your string includes one type of quote (or apostrophe), use the other one:



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

Strings

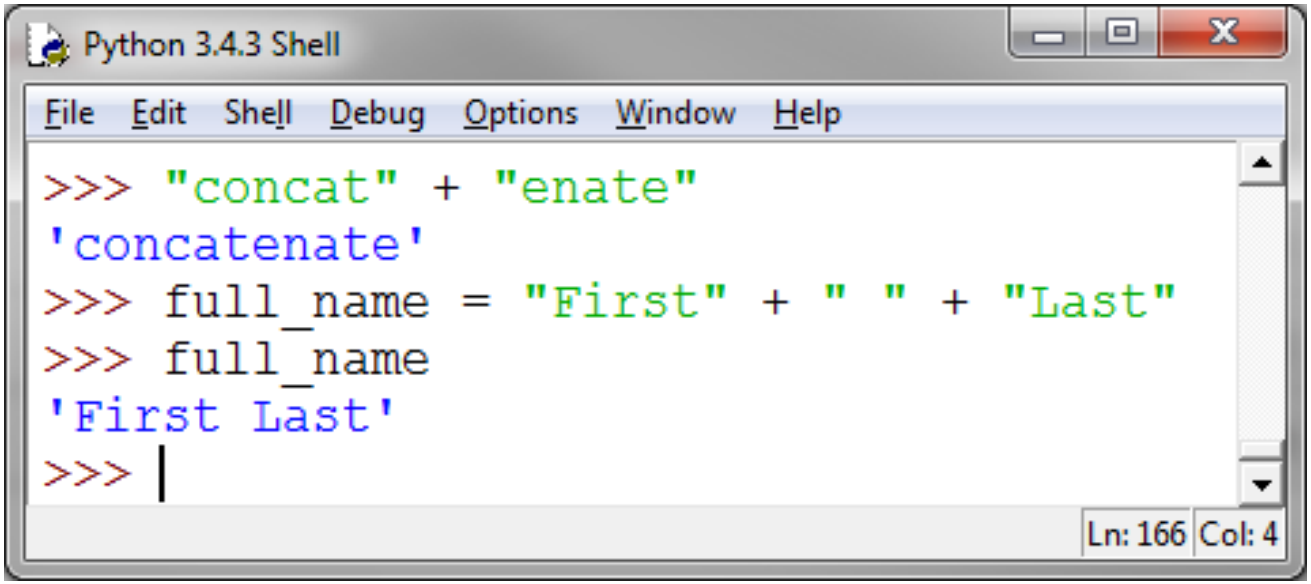
Strings can be stored in variables also:



String Concatenating

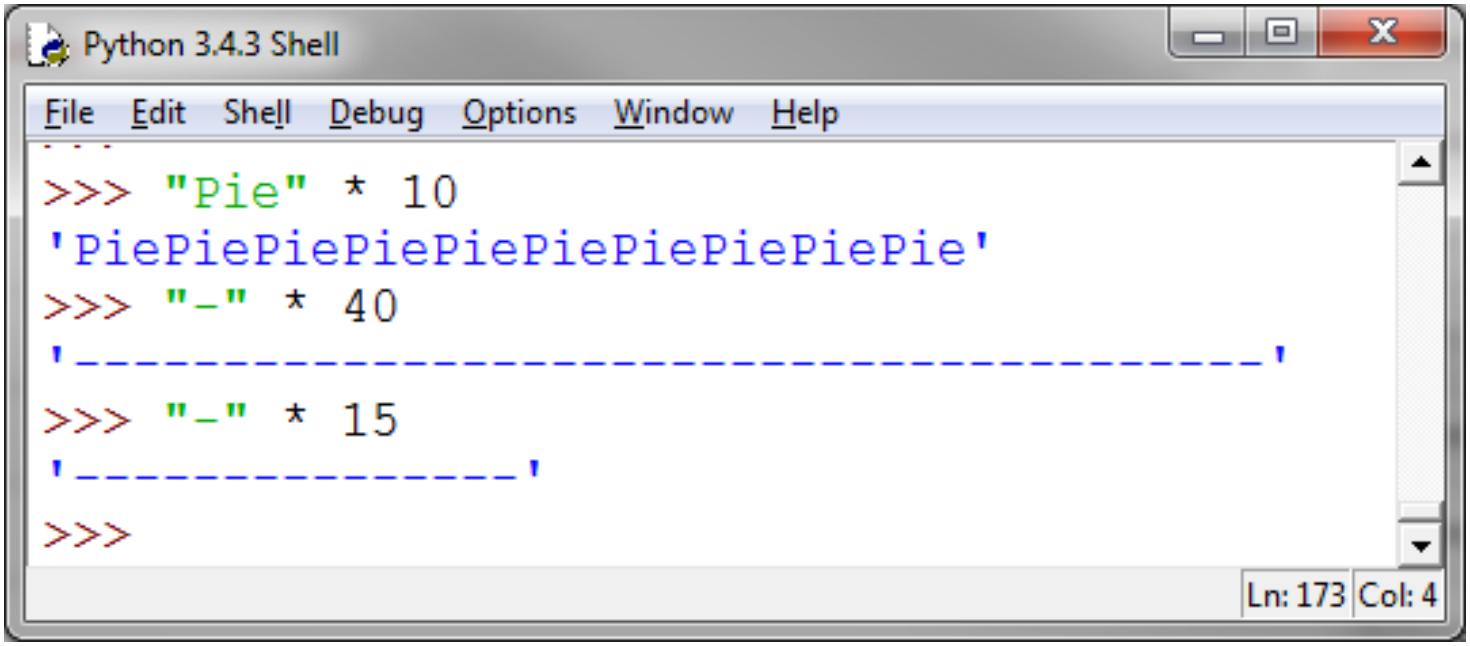
Concatenation: Joining together of two pieces to form a larger whole

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



Repeating Strings

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

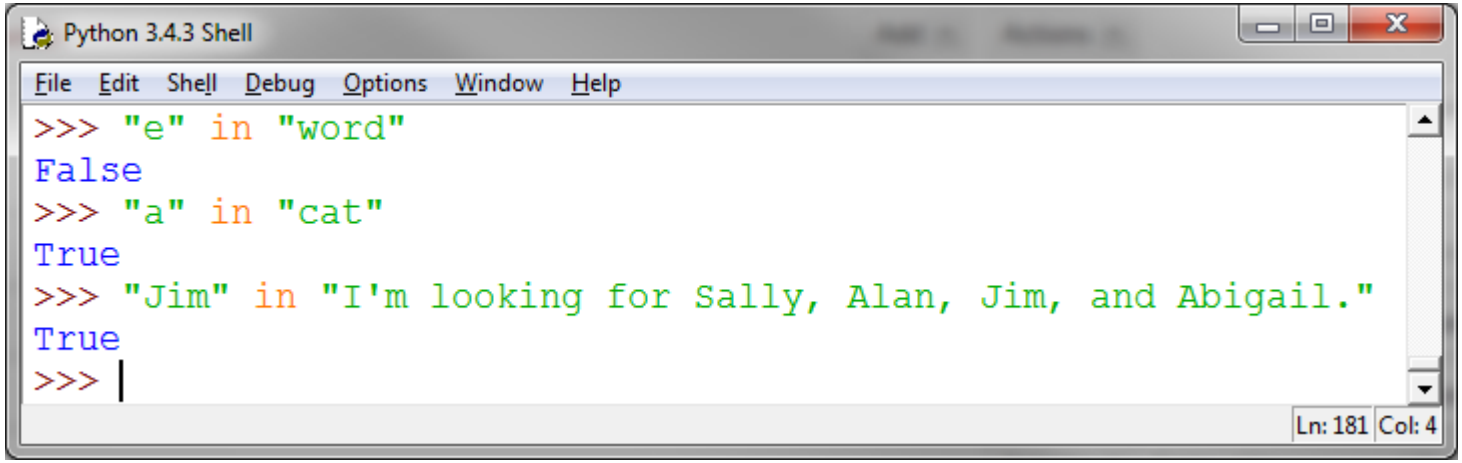


This will be useful later!

in operator

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

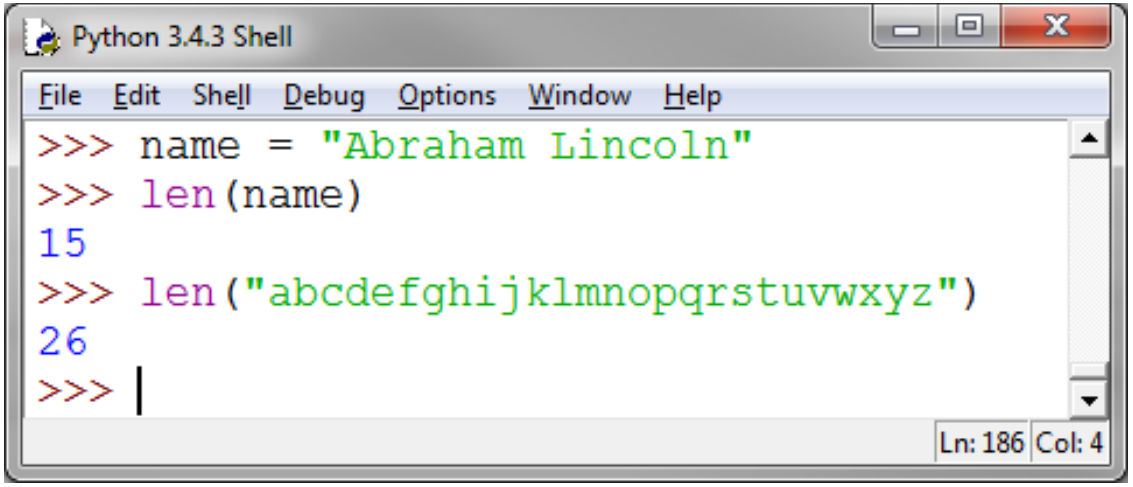
- We can test for a string being a **substring** of another string:



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:



Indexing strings

word = "learn”

L	E	A	R	N
0	1	2	3	4

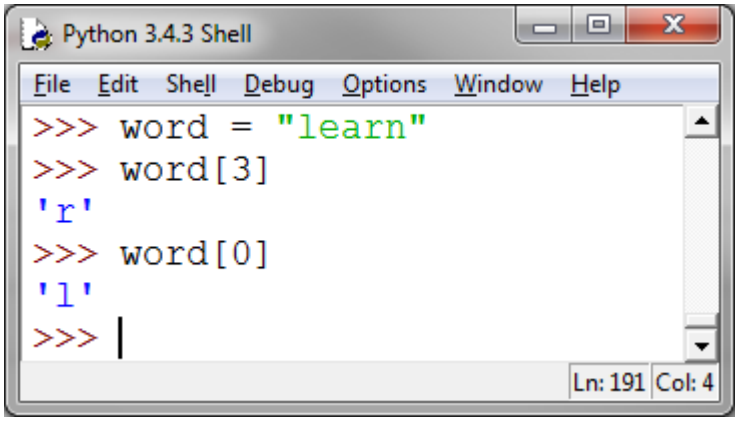
- Each letter in a string has a numeric position called an **index**.
- This numbering starts at **0**, not **1**!!

Indexing strings

word = "learn"

L	E	A	R	N
0	1	2	3	4

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

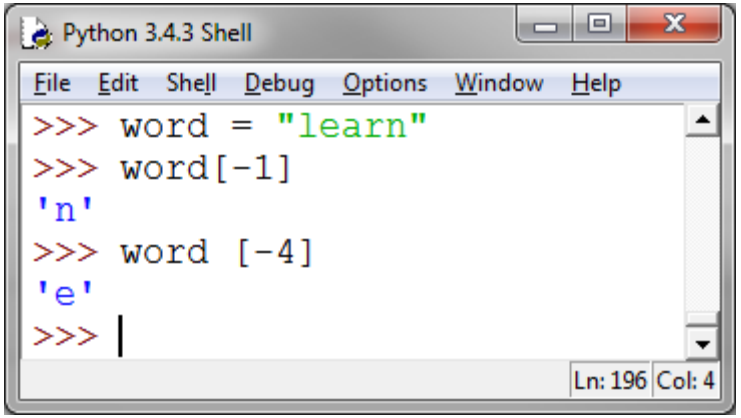


This is also called **indexing into** something.

Negative Indexing

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

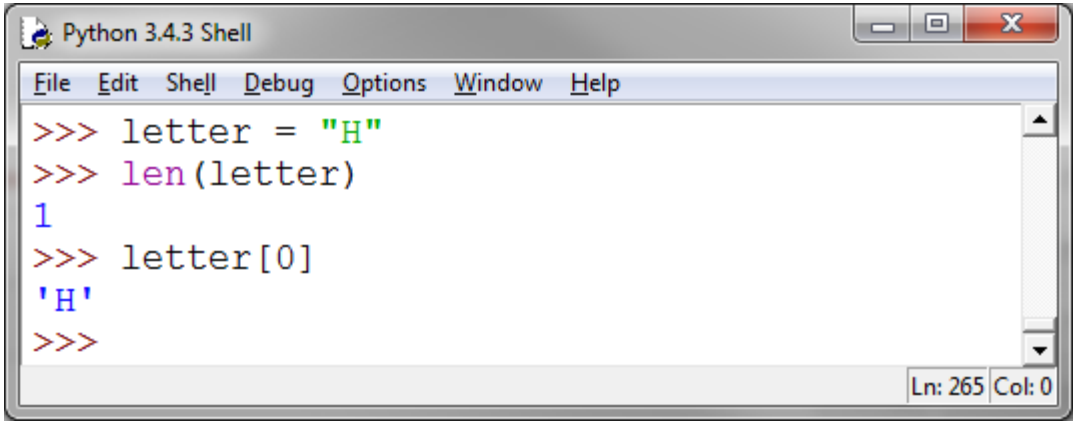
- Sometimes, we care more about the *end of a string*.
- We can use **negative indexing** to quickly refer to the characters in a string, starting at the end:



len()

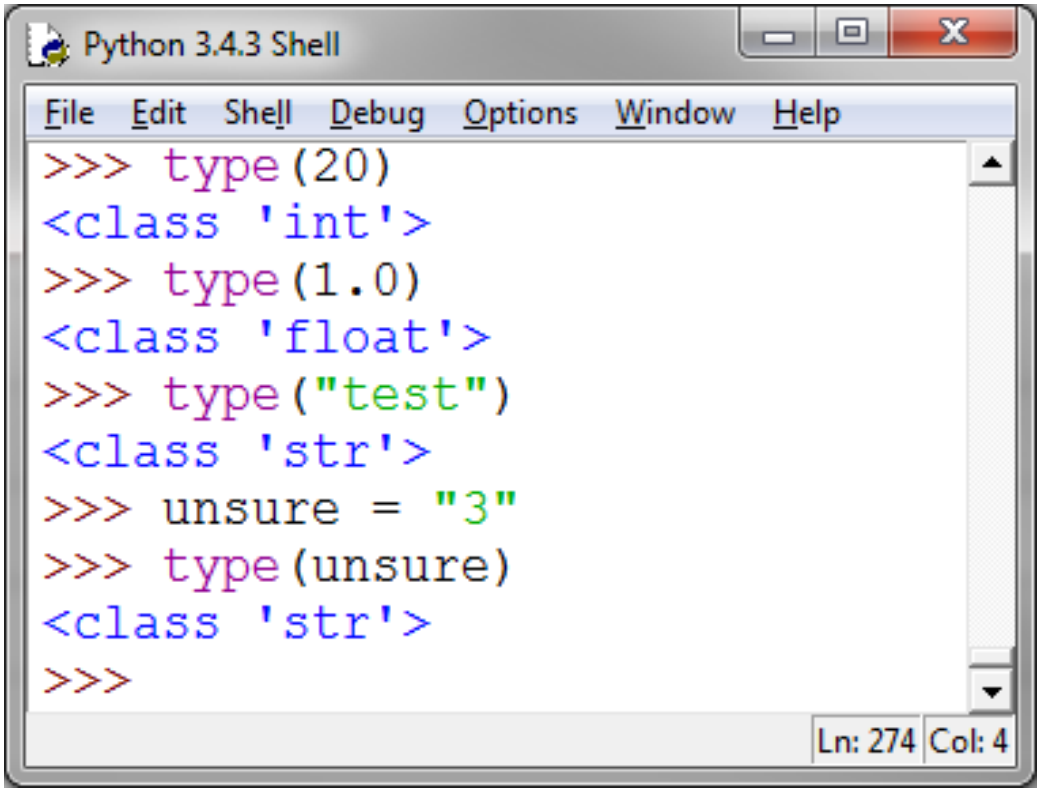
```
>>> len( "I like Python." )
14
```

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



The type function

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



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*

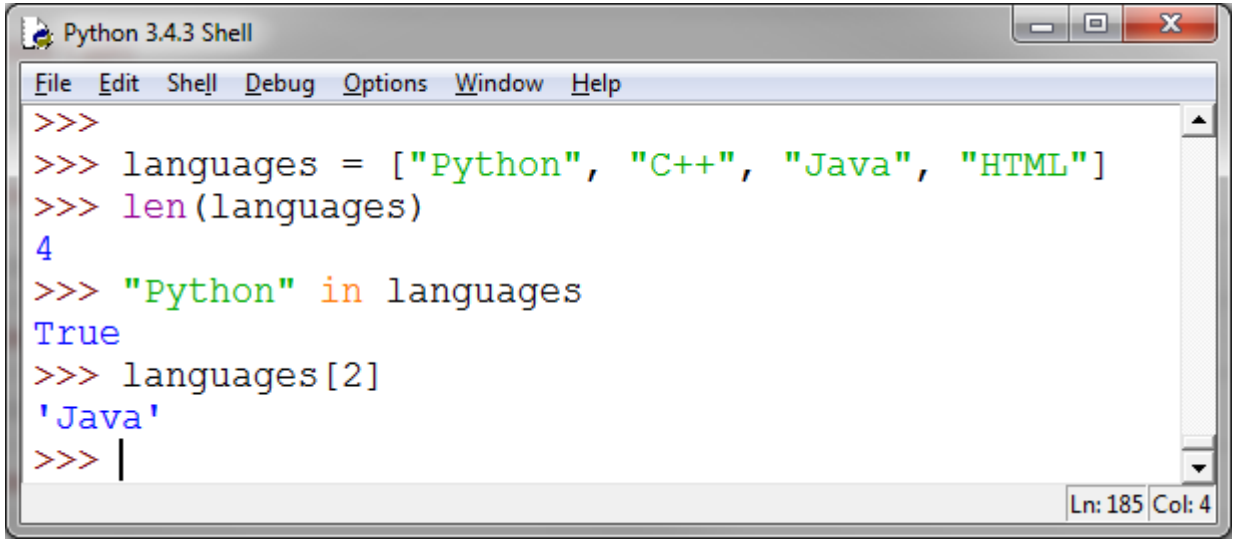
Example: The 10 best movies this year

- This 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:



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:

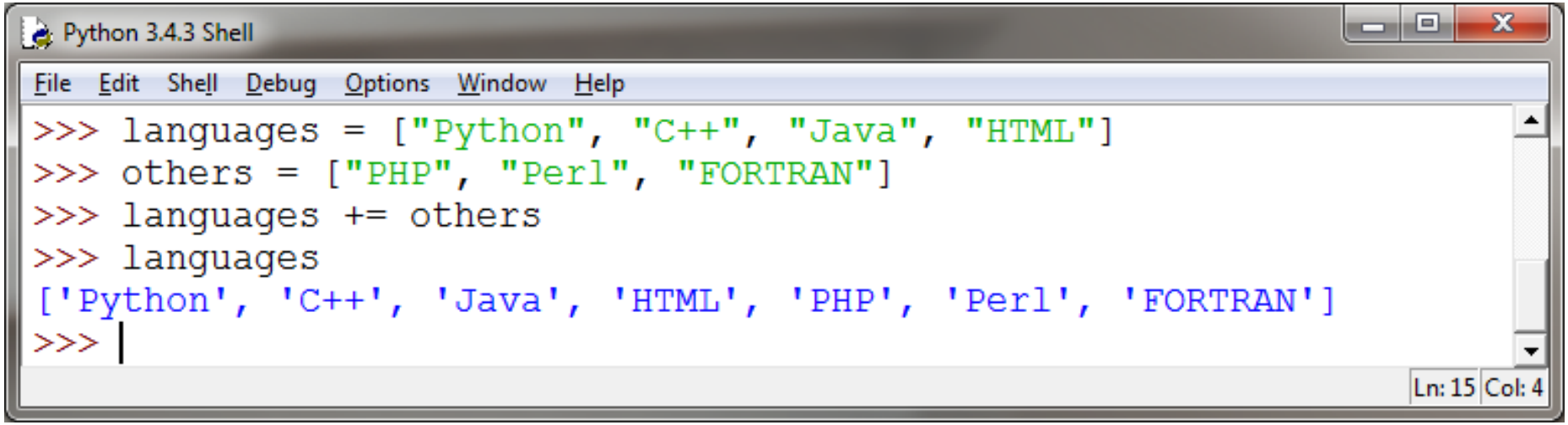
Original: **score = score + 1**

Augmented: **score += 1**

Operator	Example	Is Equivalent To
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 2
%=	x %= 5	x = x % 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5

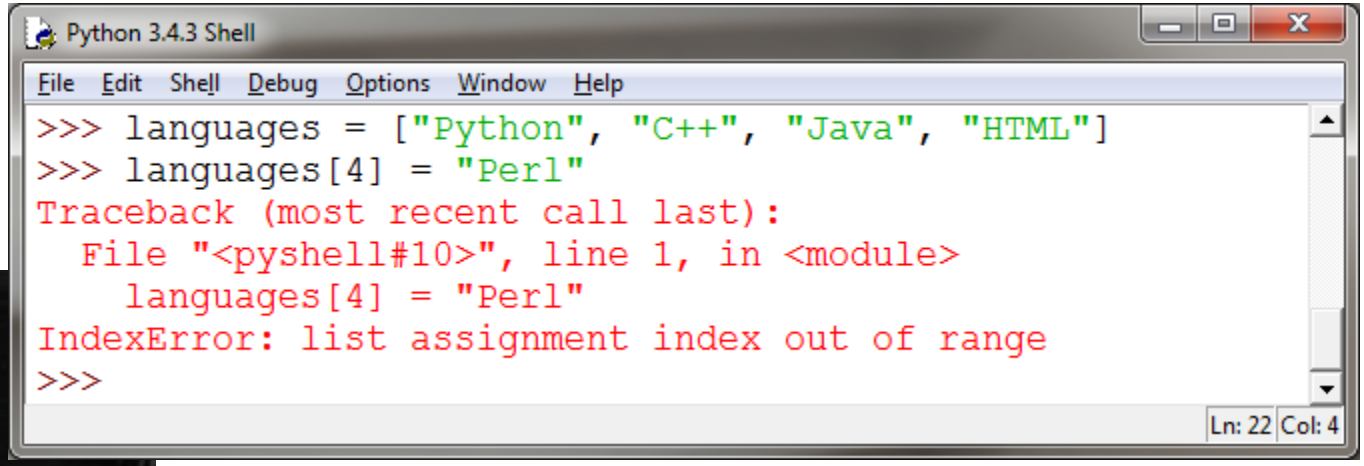
Lists – Concatenating (adding)

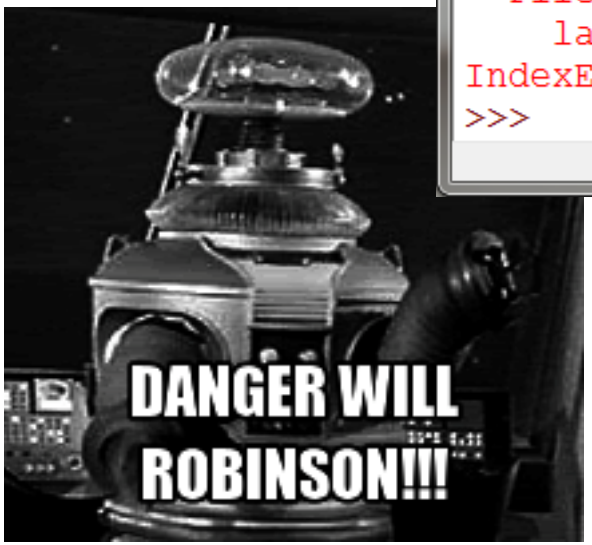
Lists **add** like strings and tuples:



Lists – Changing Elements

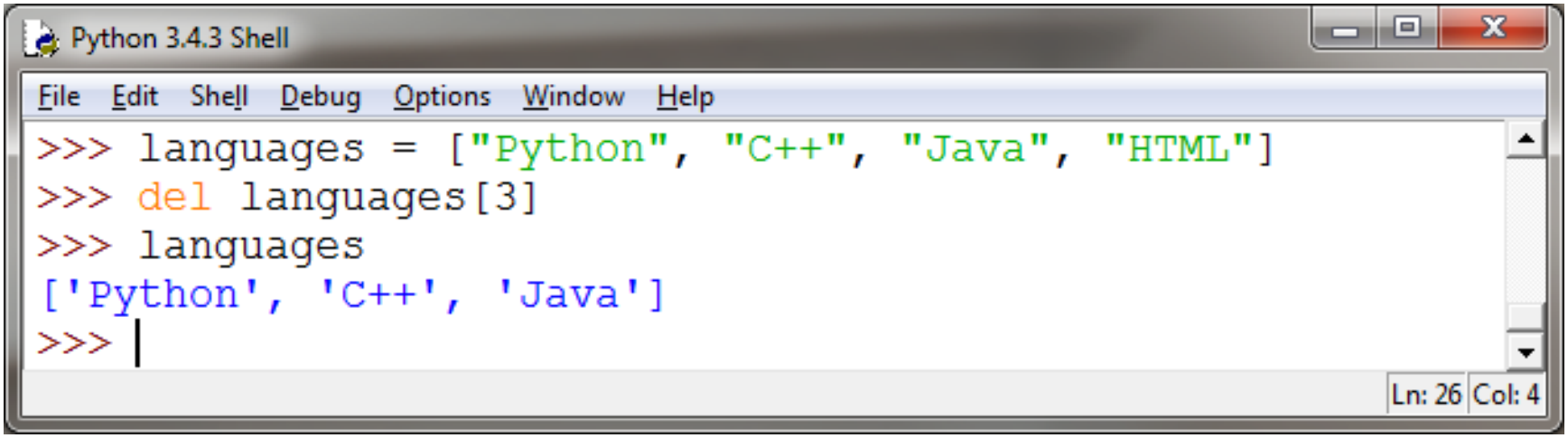
Make sure you don't try to change elements that don't exist!!





Lists – Deleting Elements

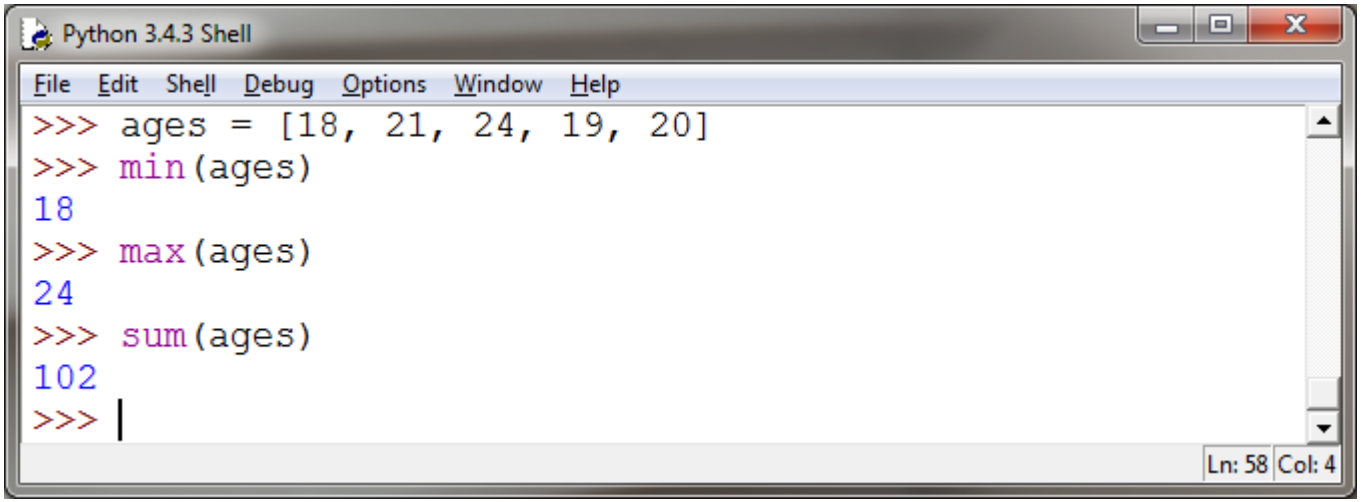
We can erase a portion of the list by using the keyword **del**:



del

Lists – min, max, sum

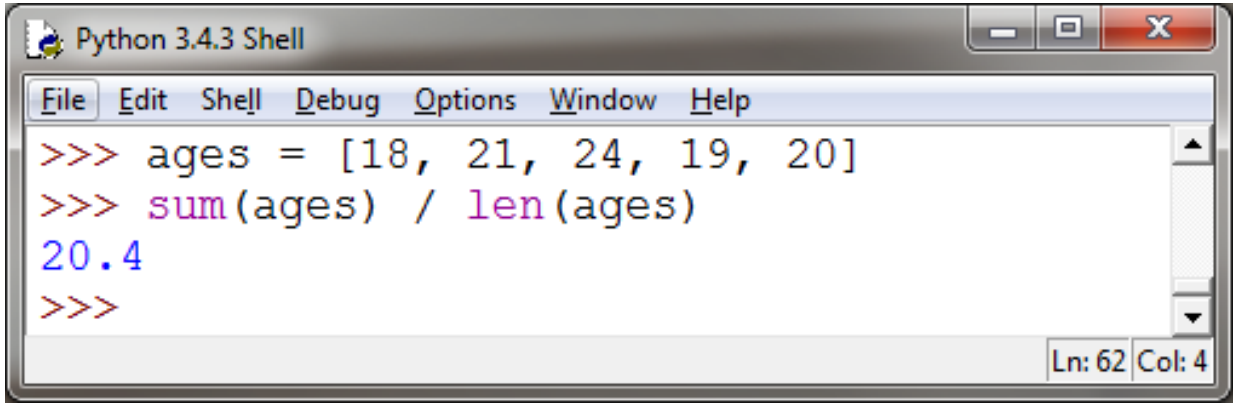
Python has three very useful functions for working with numeric lists:



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:



- 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:

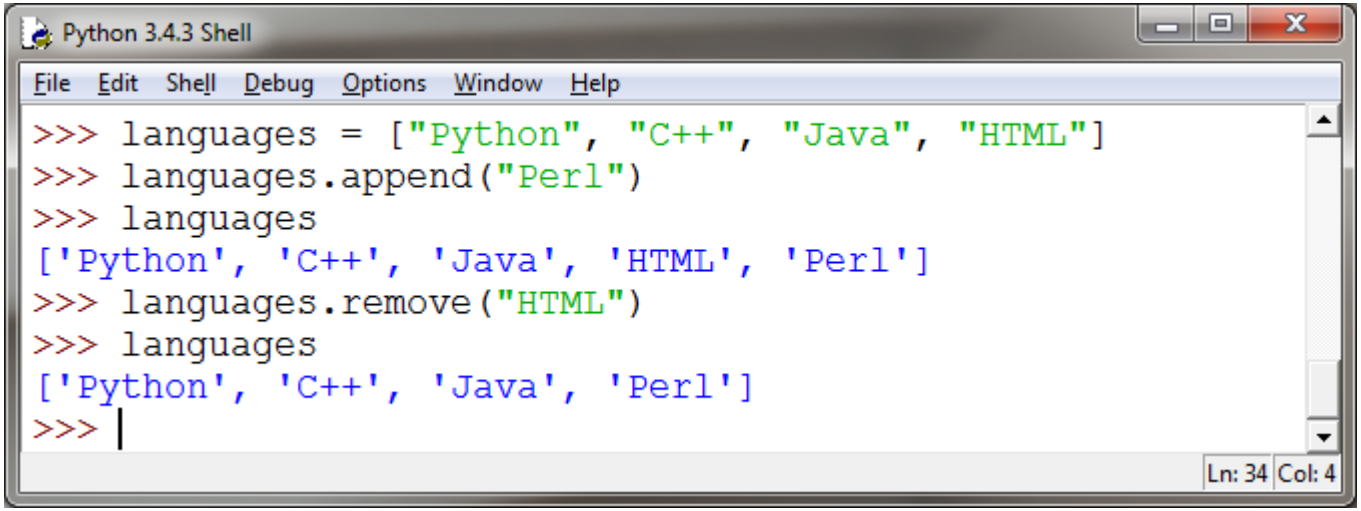
```
object_name.method_name()
```

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

```
>>> float(3)
3.0
```

Lists – Appending and Removing

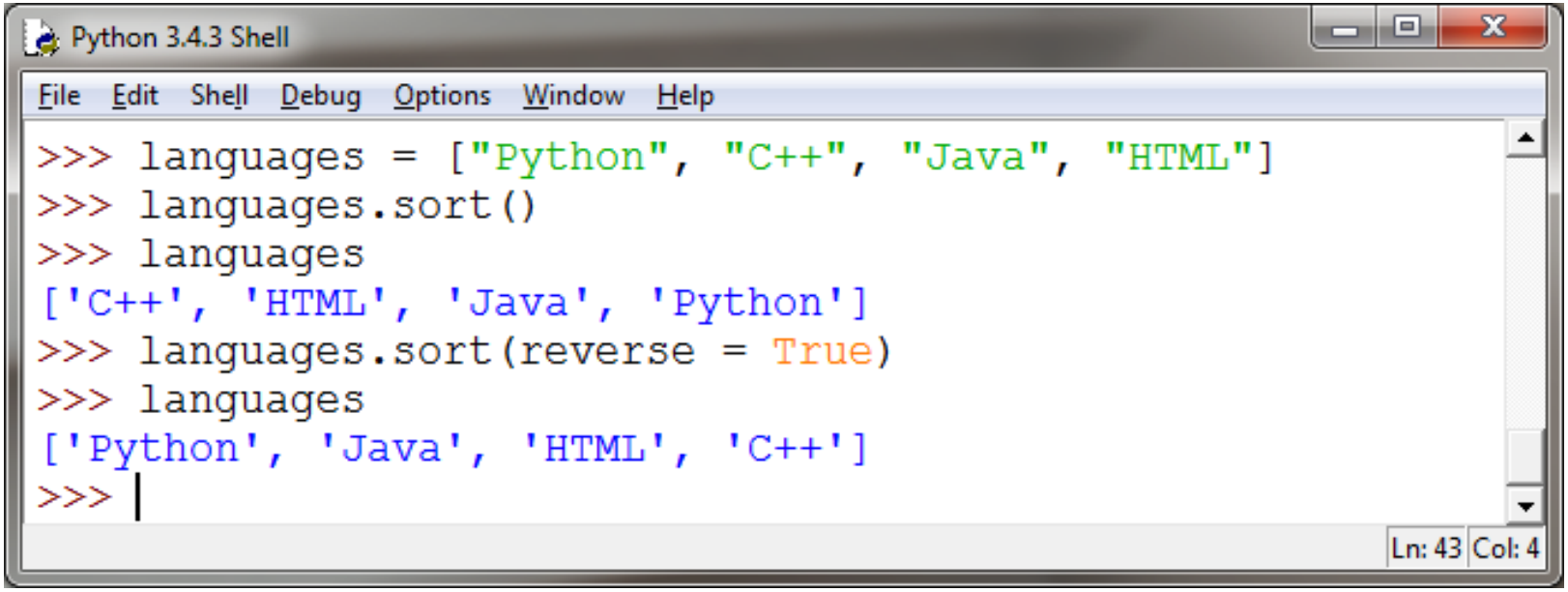
The **.append()** method is the preferred way to add things to a list! It will add the new element to the end.



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

Lists - Sorting

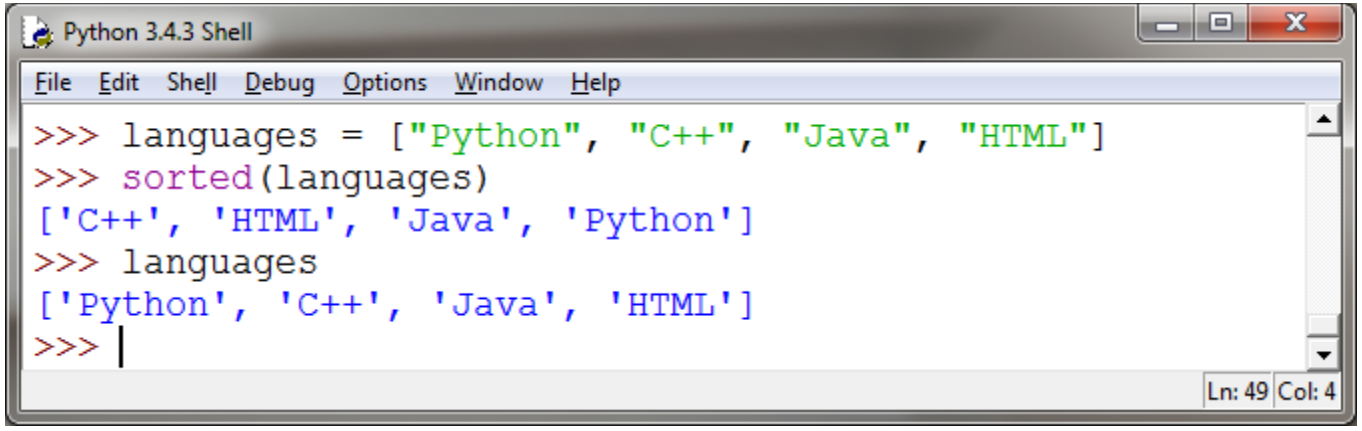
We can **.sort()** a list in ascending or descending order:



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

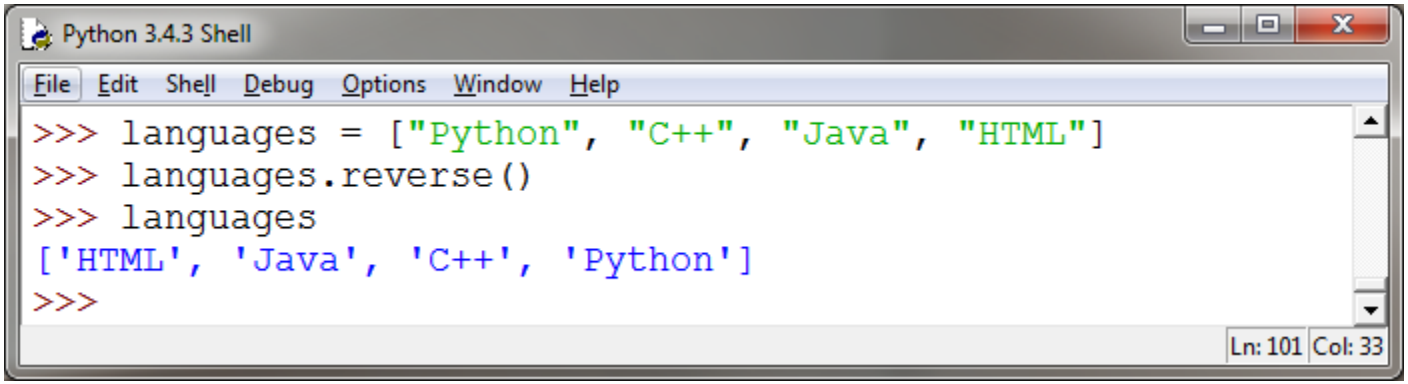
Lists - Sorting

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.



Lists - Reverse

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



Questions?

Lecture 4

I210 – INTRODUCTION TO PROGRAMMING WITH PYTHON

Today

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

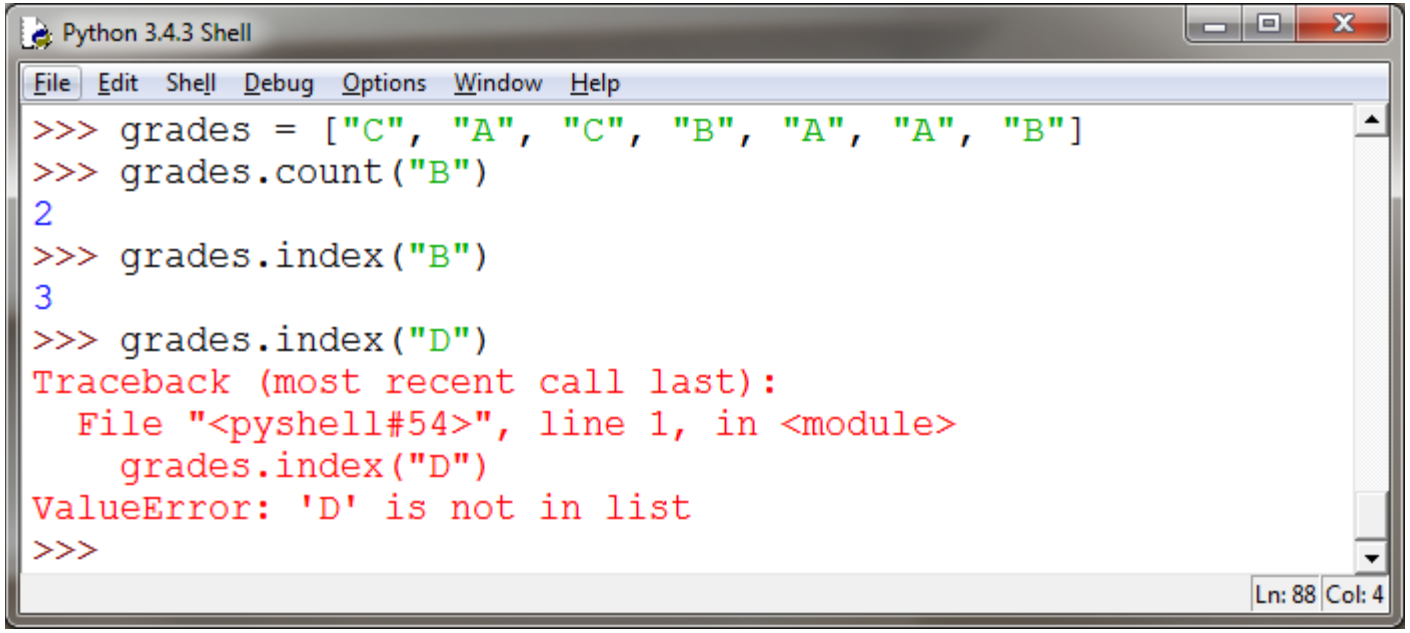
.join() .sort() copy() dir() help() input()

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.

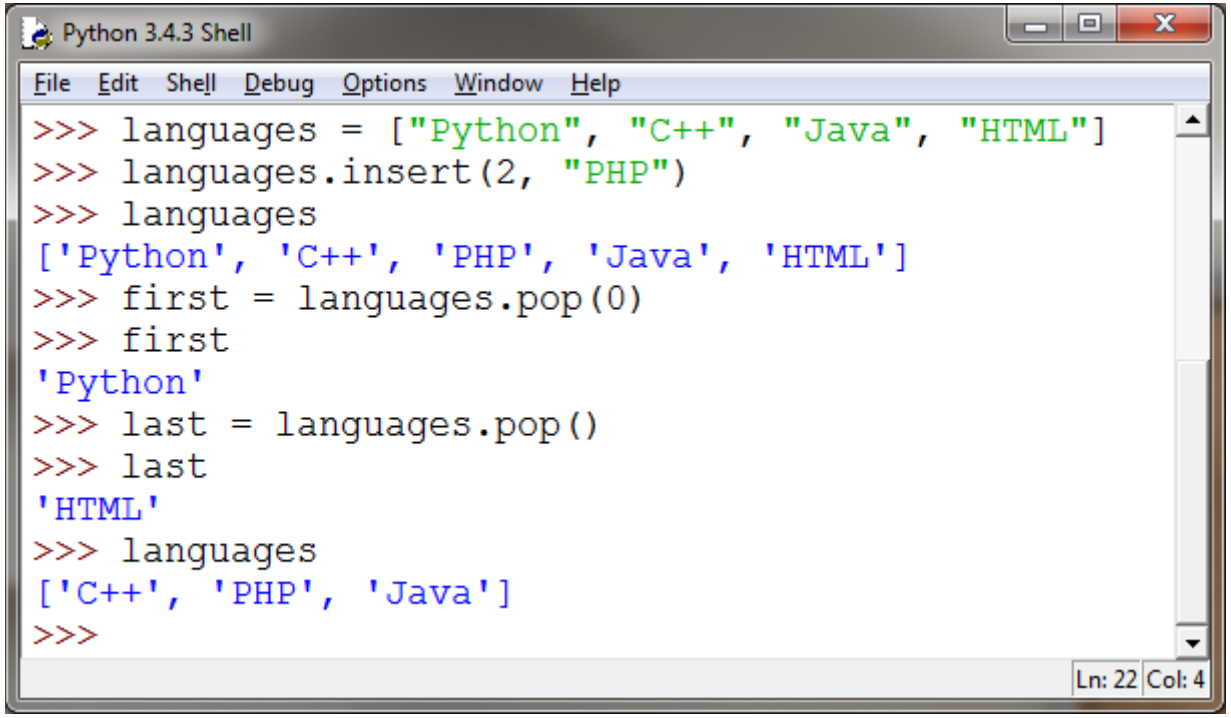
Caution! You'll get an error if your target isn't in the list!



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.

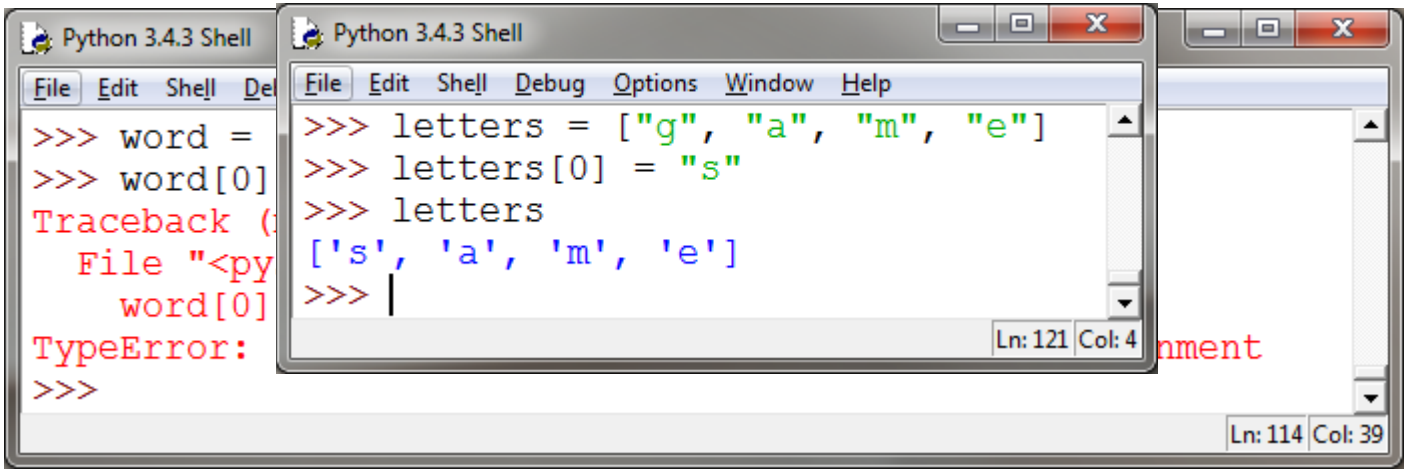


Immutable vs Mutable Sequences

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.

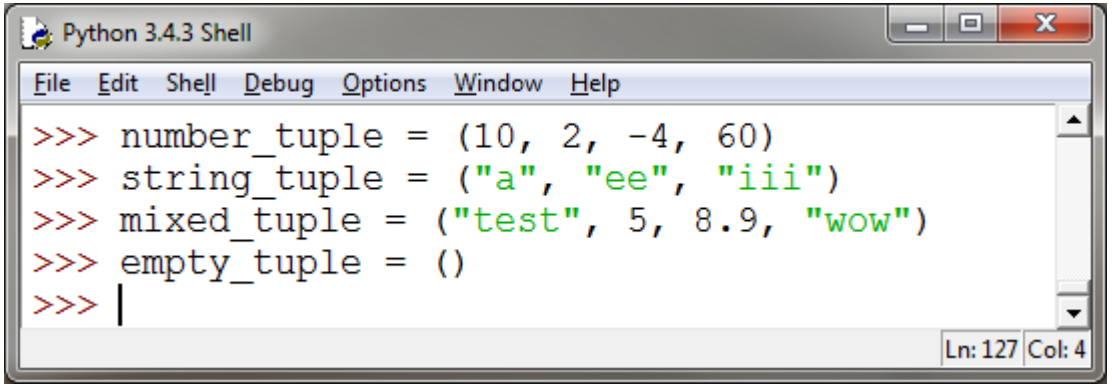


Tuples

Tuples are essentially immutable lists.

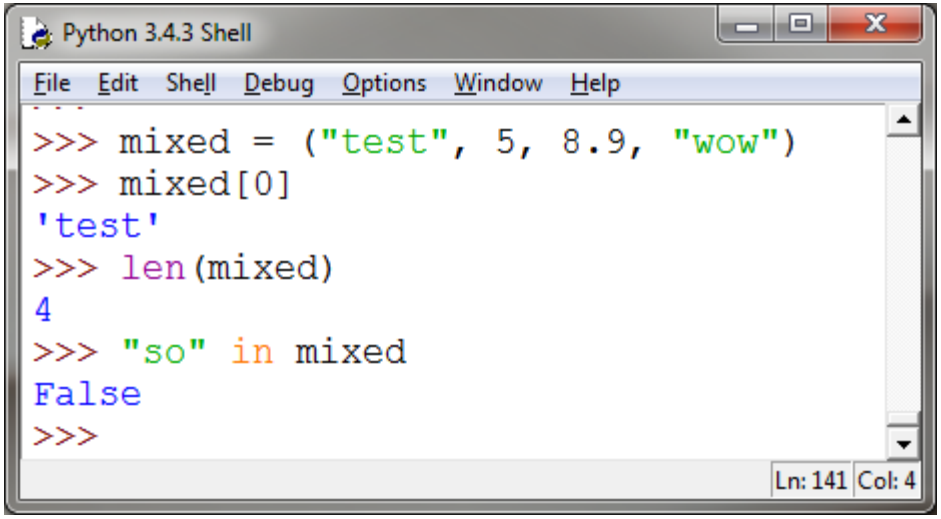
– (Tuples are also known as **arrays**.)

- We create tuples with parenthesis: **( )**



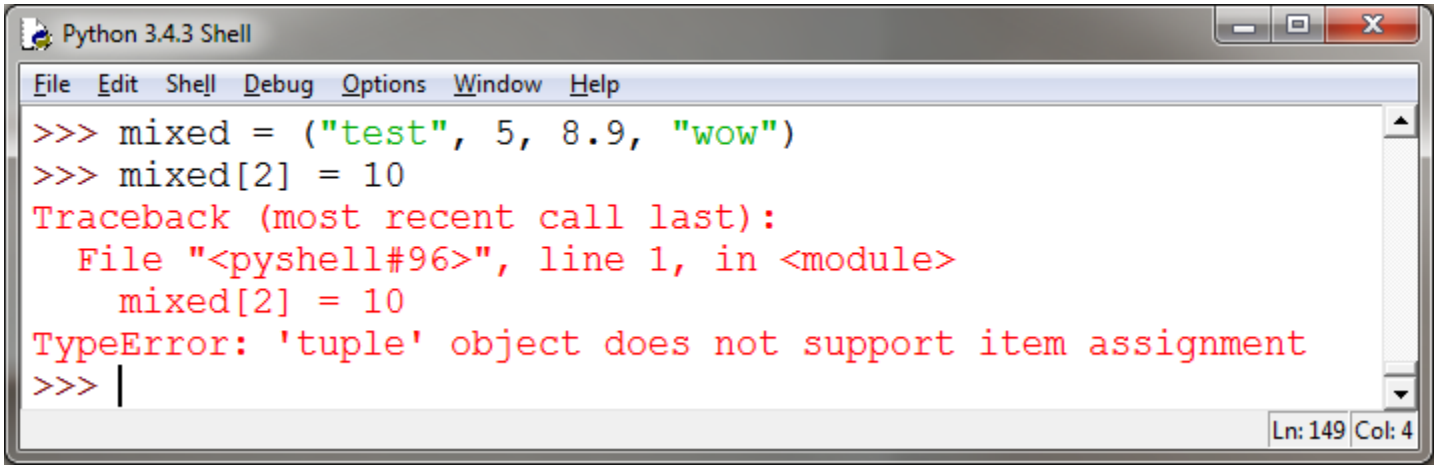
Tuples

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



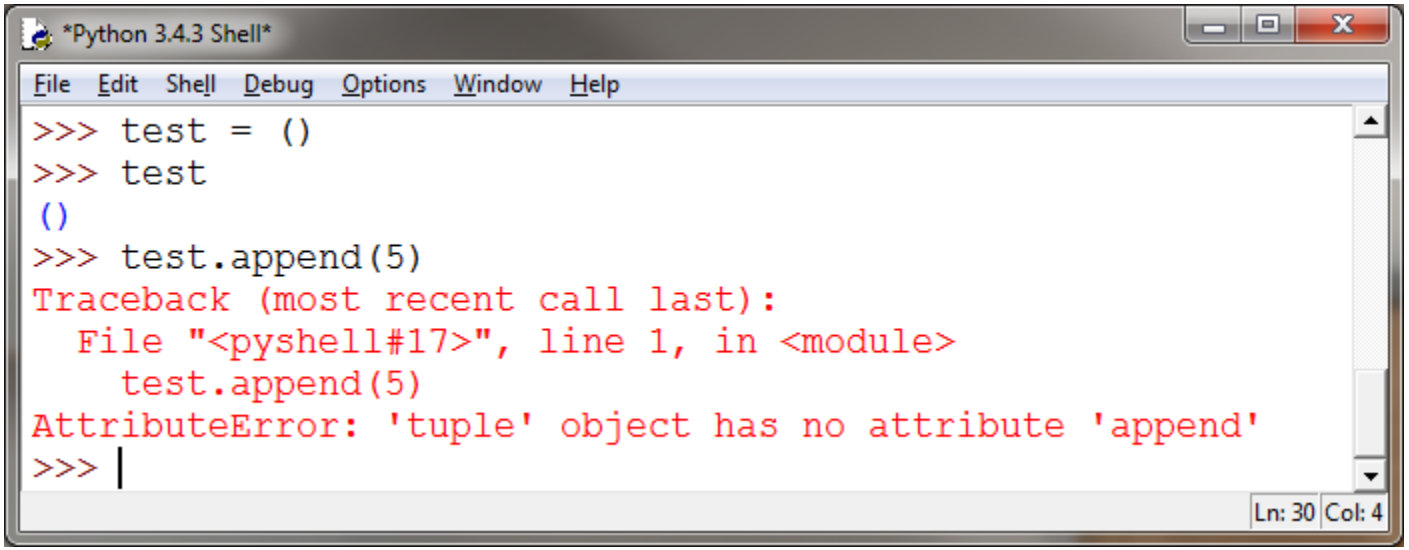
Tuples

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



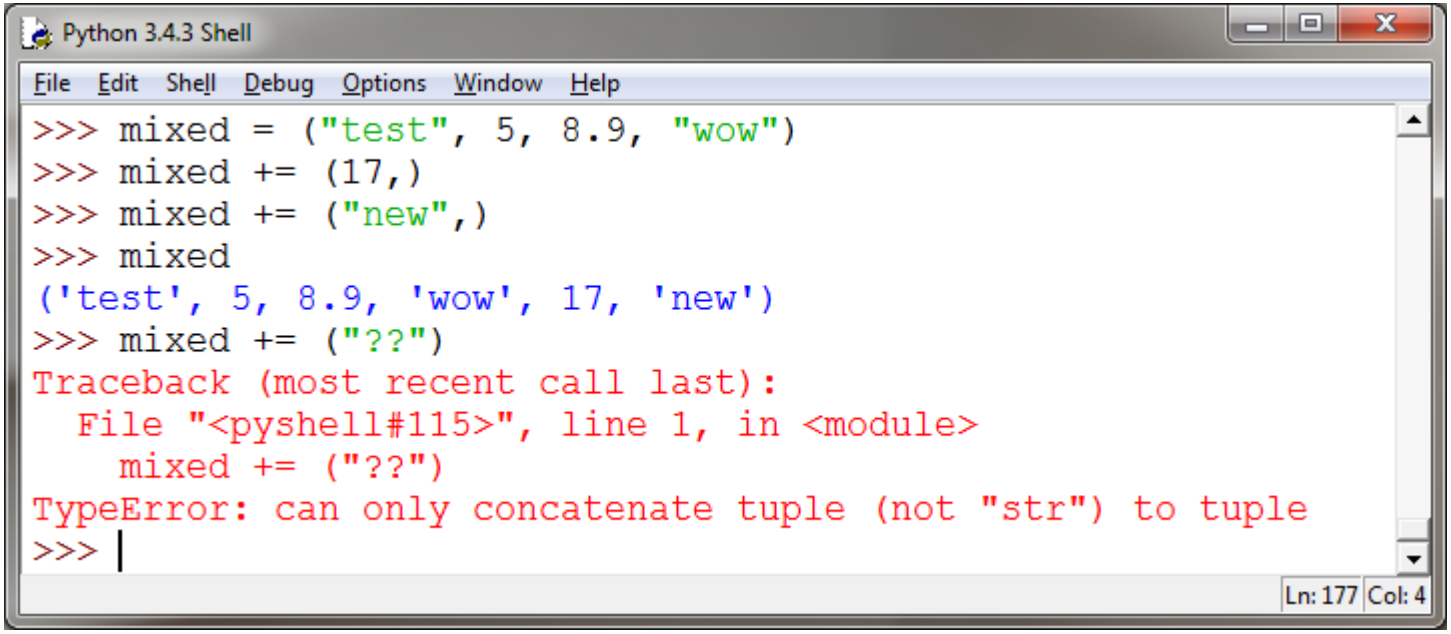
Tuples

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



Tuples

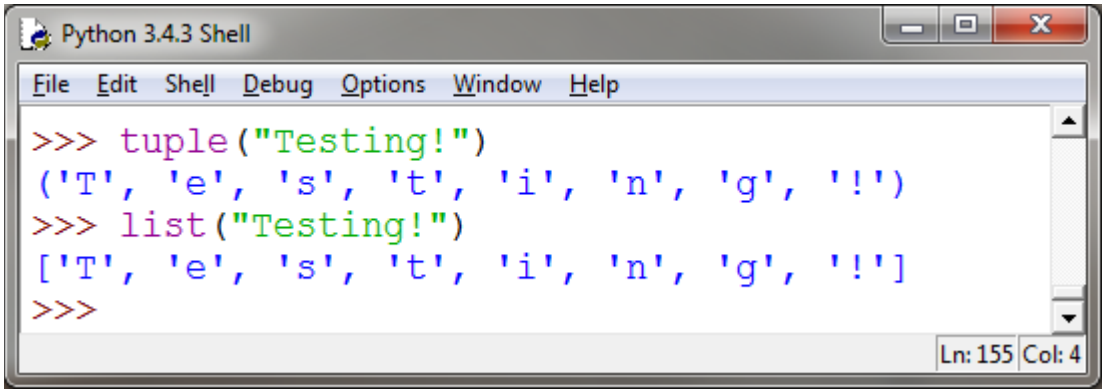
… but we can add elements to tuples with **concatenation!**



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

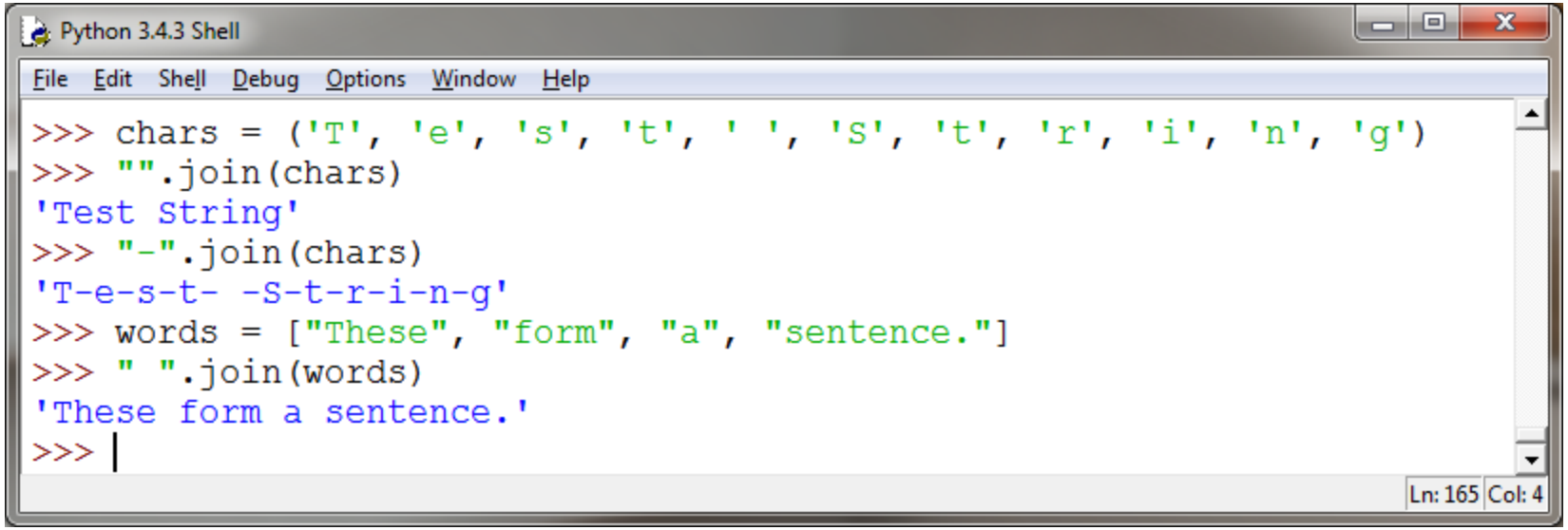
The tuple() and list() Functions

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



.join

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

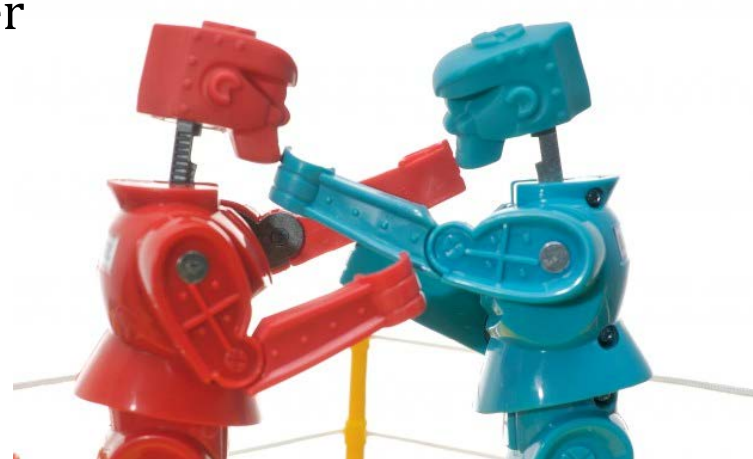


Tuples vs. Lists – Why have both?

- 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.

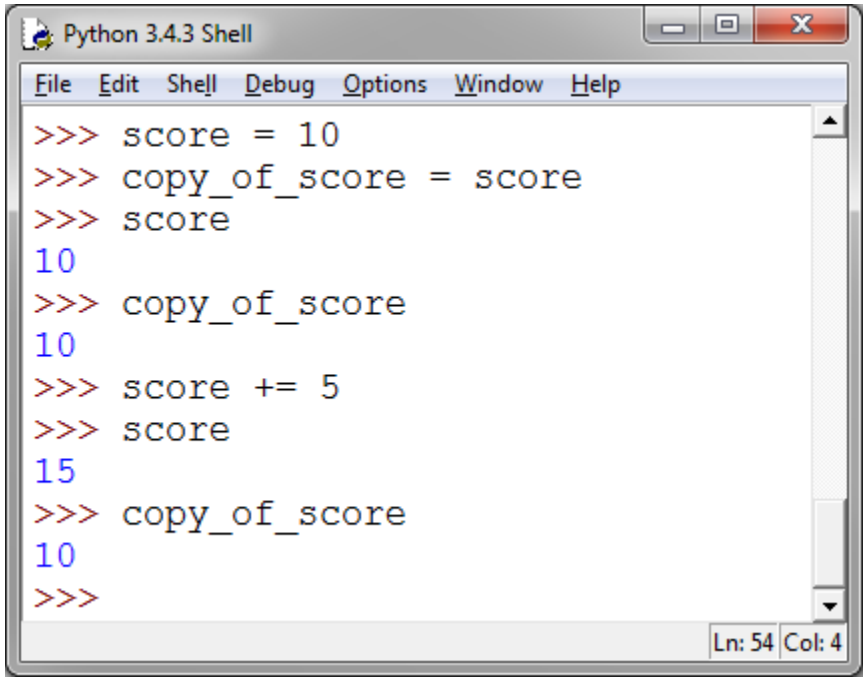
Classic trade-off in Programming: **convenience** *vs.* **speed**

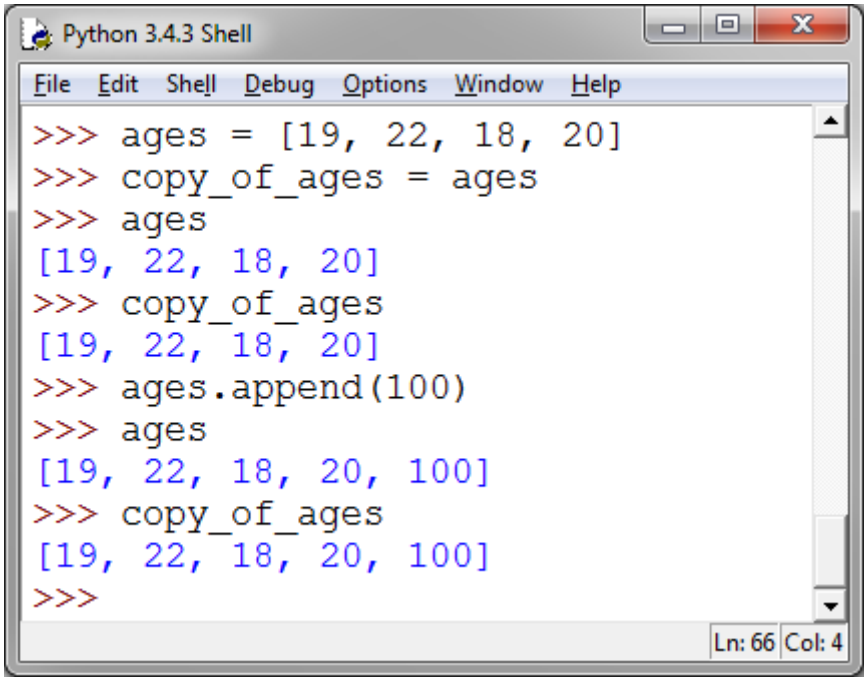
- Assuming that elements can't change makes tuples much faster, but programming harder
- So Python gives you two options:
  - **tuples (fast but limited)**
  - **lists (slow but flexible)**



Shared References

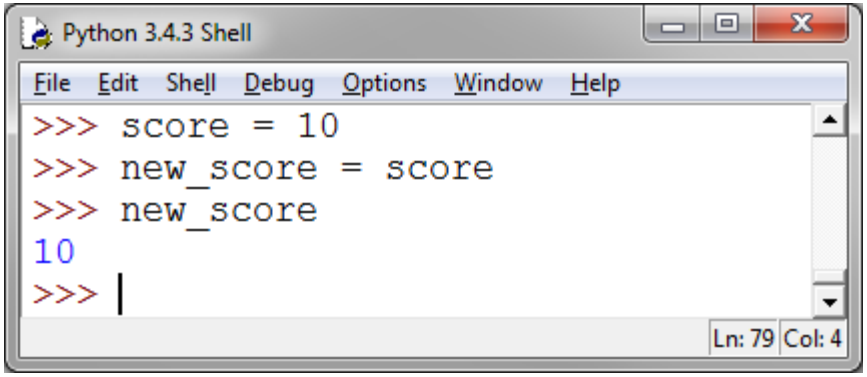
Take 1 minute to look at these two examples with your group. *Discuss what's different.*

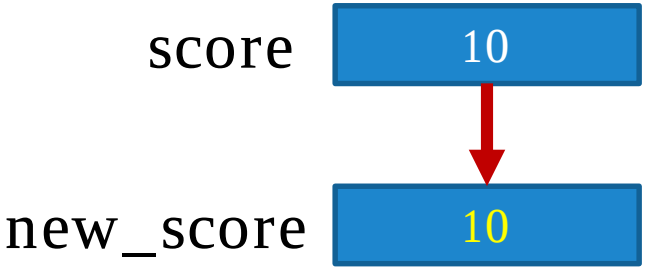




Shared References

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.

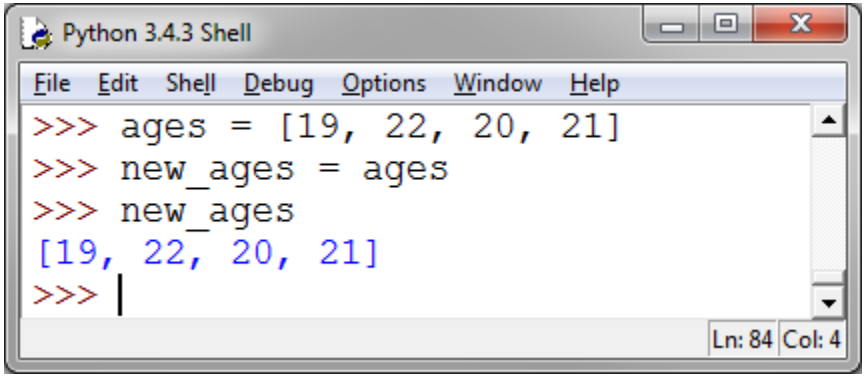


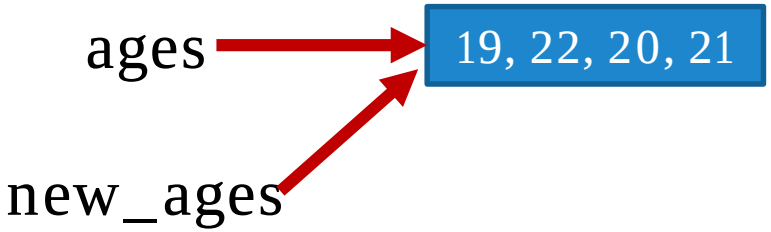


Shared References

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.

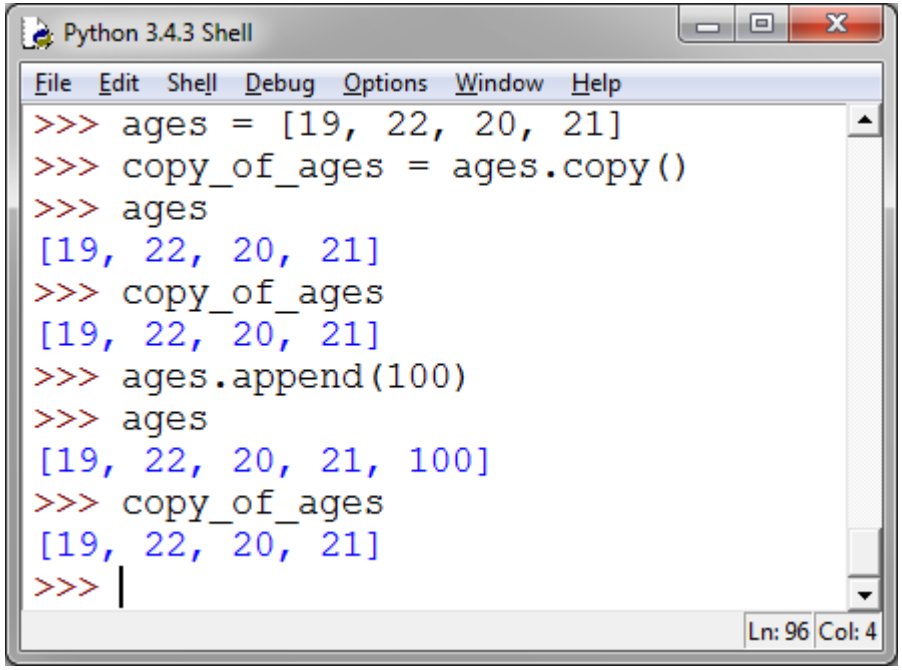
So anything we do to one is done to the other!





The .copy() Method

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



Modules

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

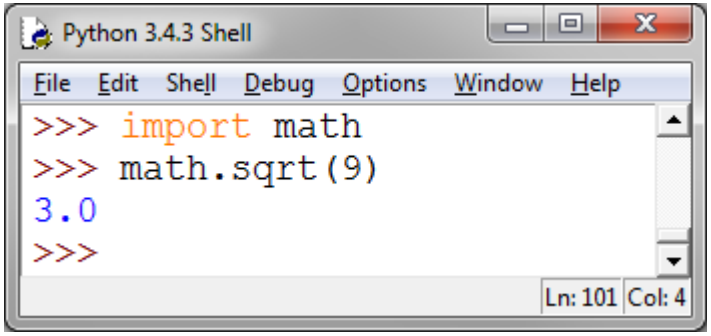
- Modules can be used by **importing** them, at which point the code becomes part of the current program.
- *Not all of Python is loaded by default!*
  Importing allows us to use more of the language.

Using Modules

Modules contain groups of related functions:

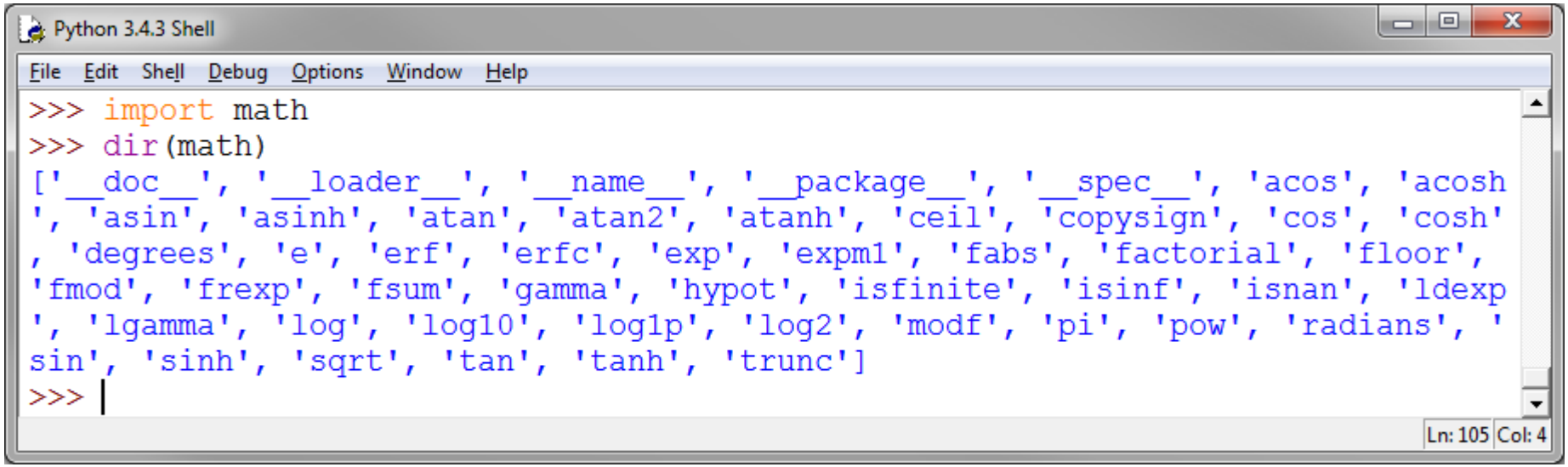
- **math** contains a lot of useful mathematical code
- Modules can also contain classes or constants
  - **math.pi**

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



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:



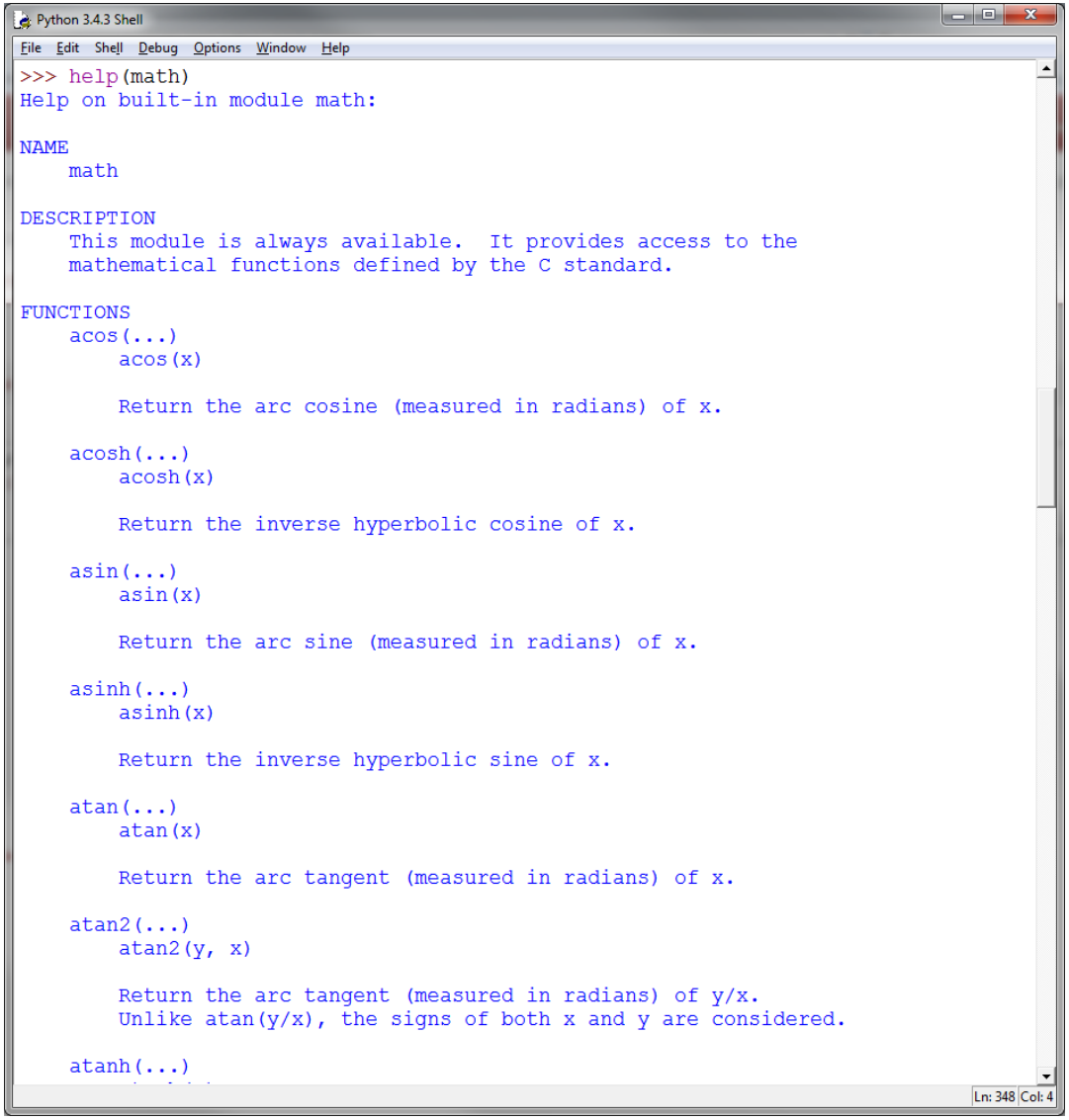
This can be helpful if you can't remember the name of a function in a module.

What's in a Module?

Need more?

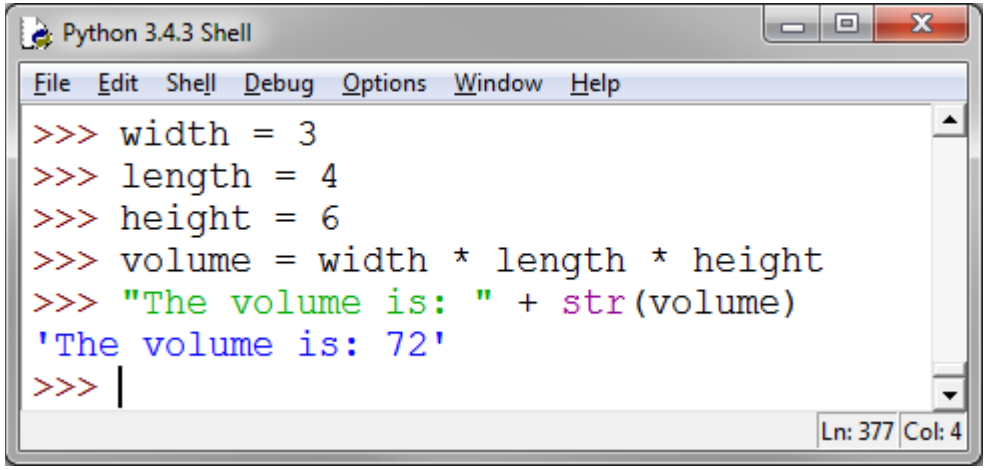
Try the **help()** function.

This gives a LOT of information.



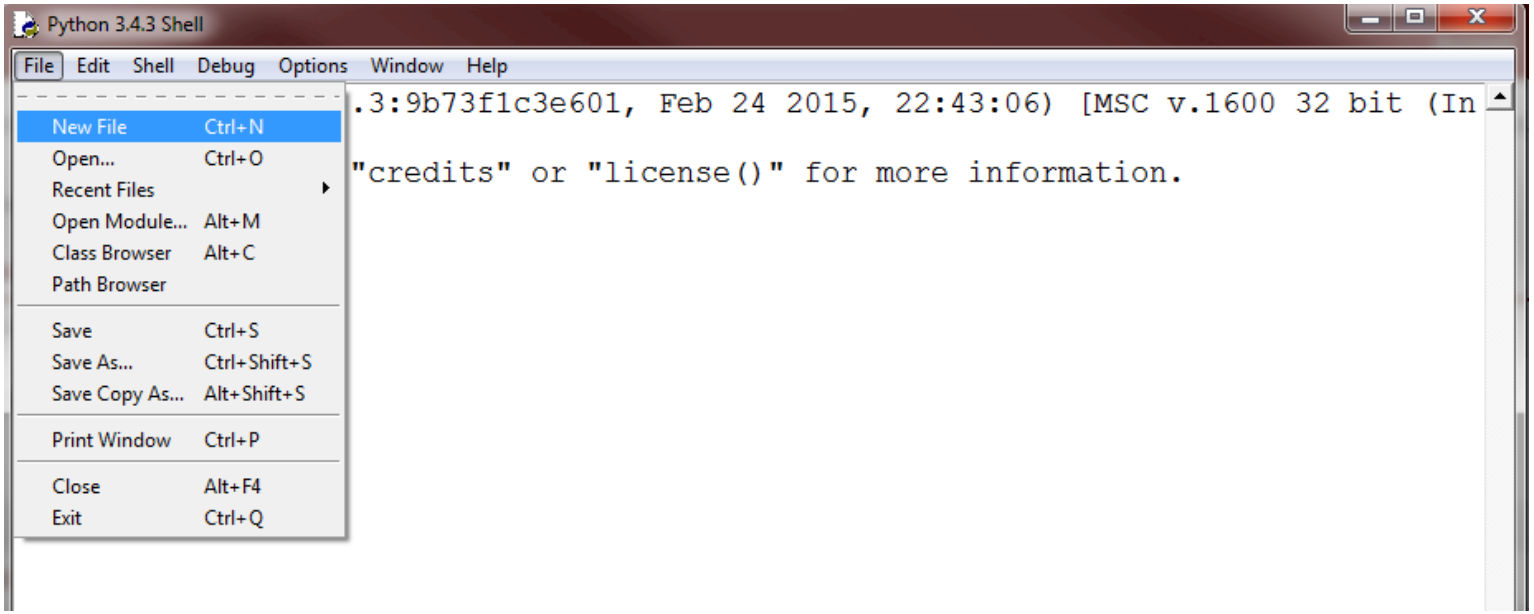
Volume (Example)

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:



Script Mode

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

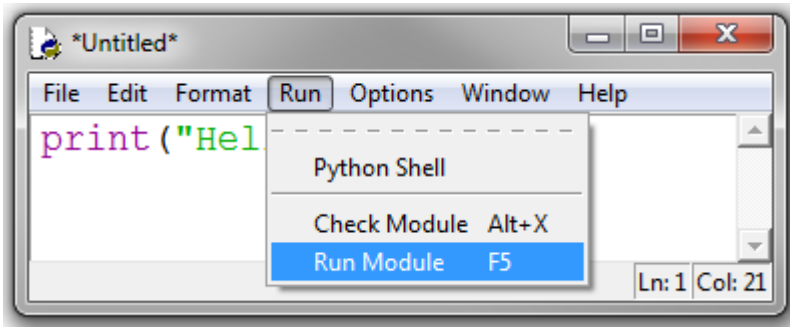


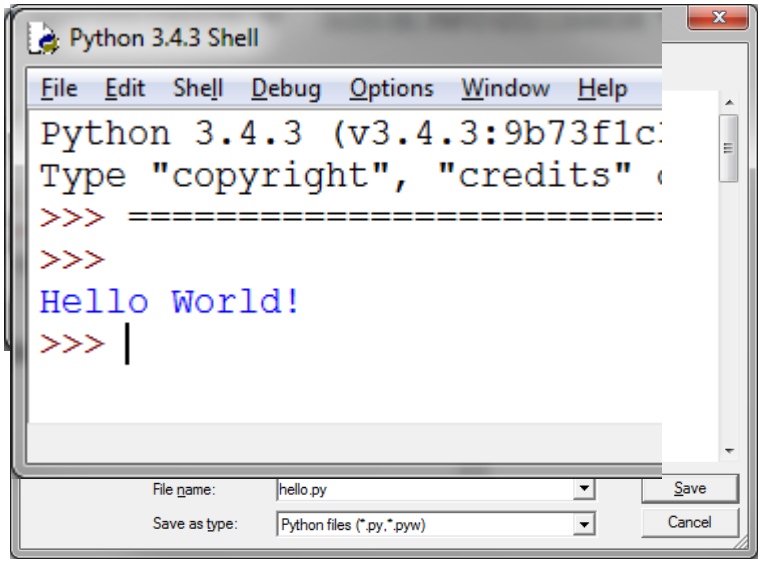
- This is where we'll write our programs from now on. We can still use Interactive Mode to test things!

The print() Function

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

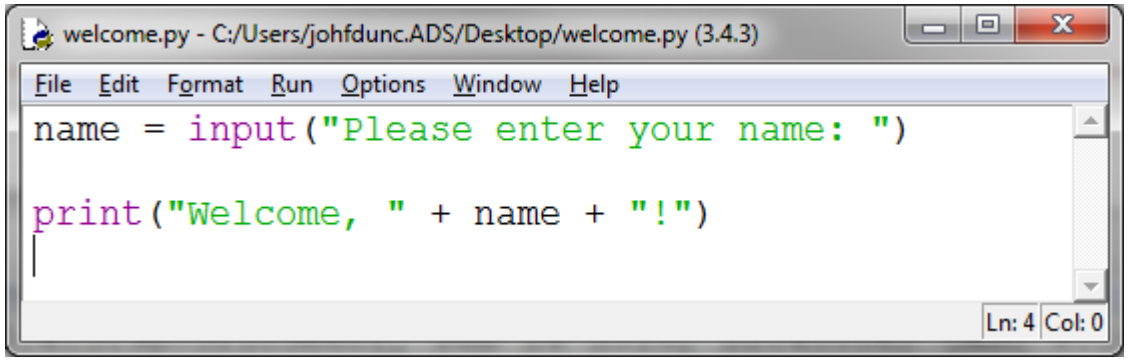
- The **print()** function takes a string and outputs it to the console. Here's a traditional first program:



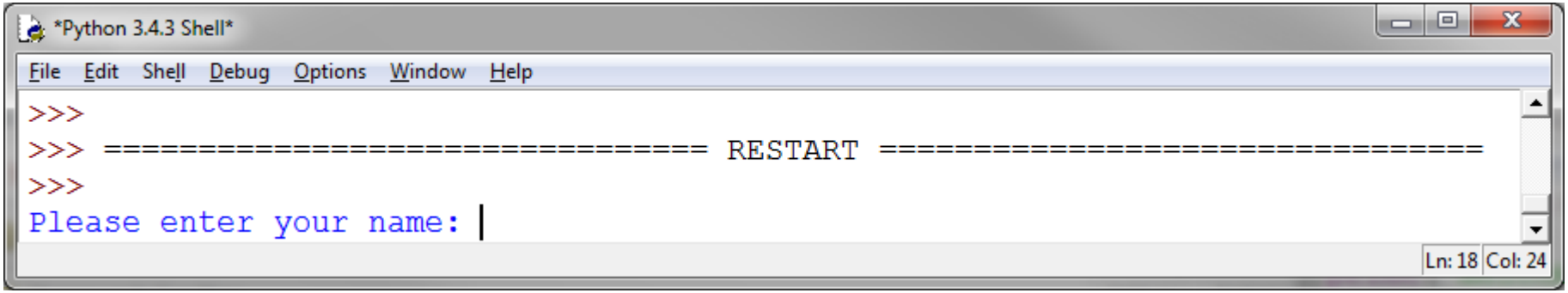


The input() Function

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

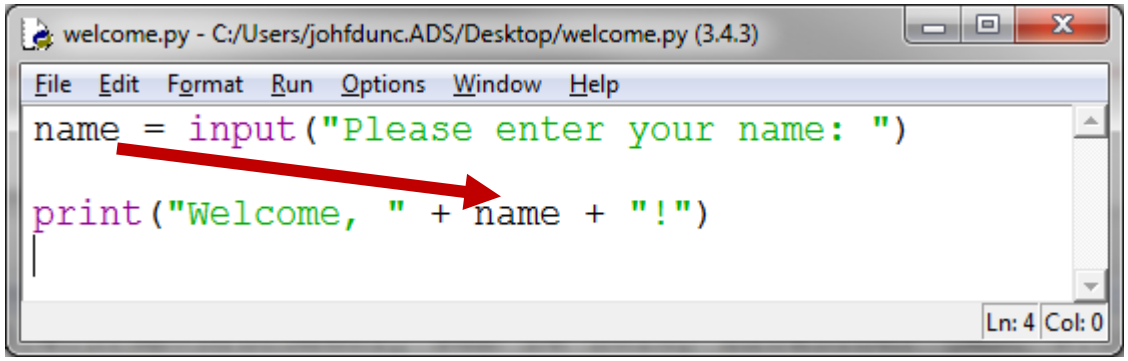


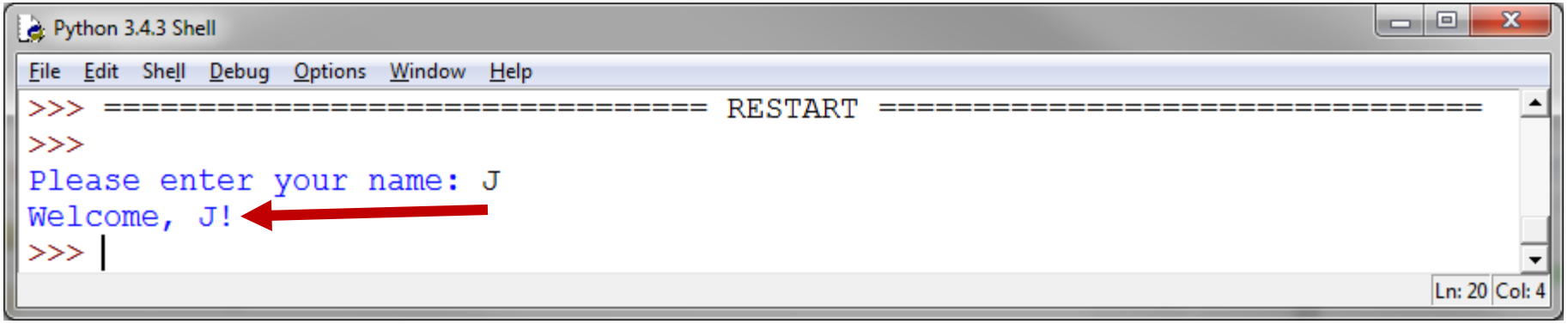
- The **input()** function prompts the user for keyboard input, and *returns a string*.



The input() Function

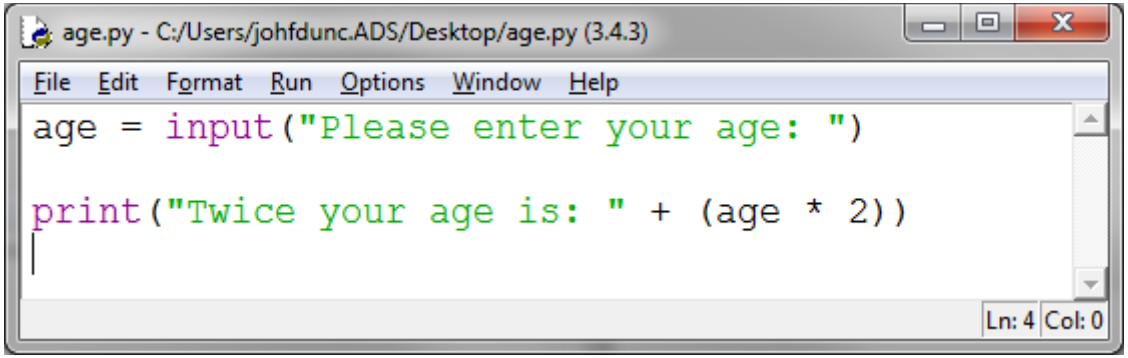
We can use information the user gives us in our programs!

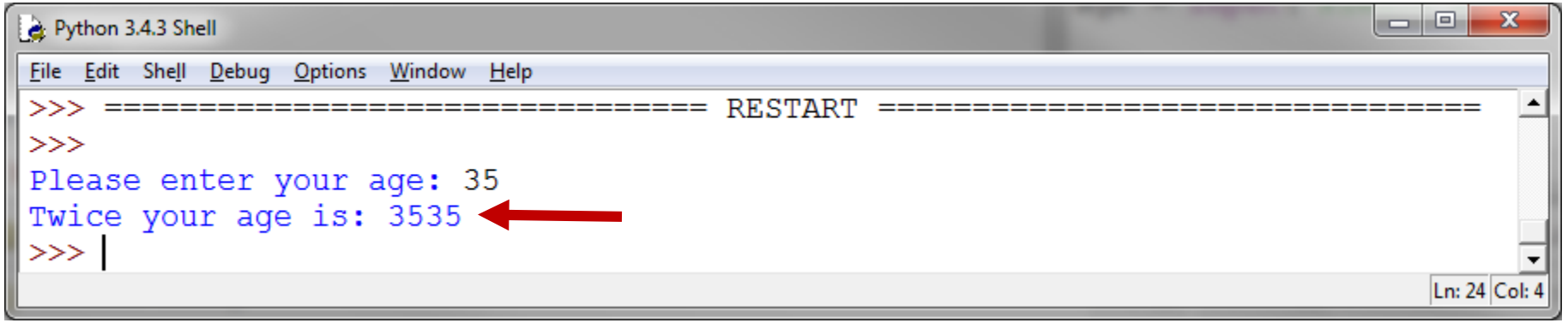




Using Data From the User

What went wrong here?

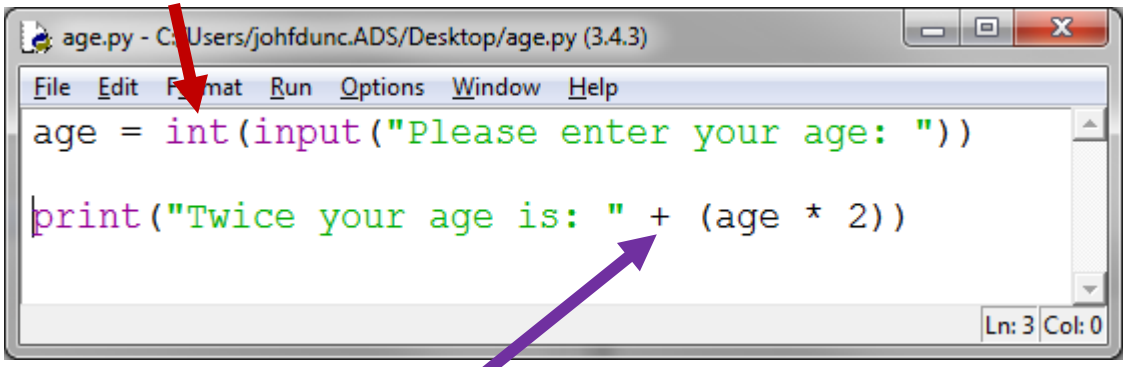


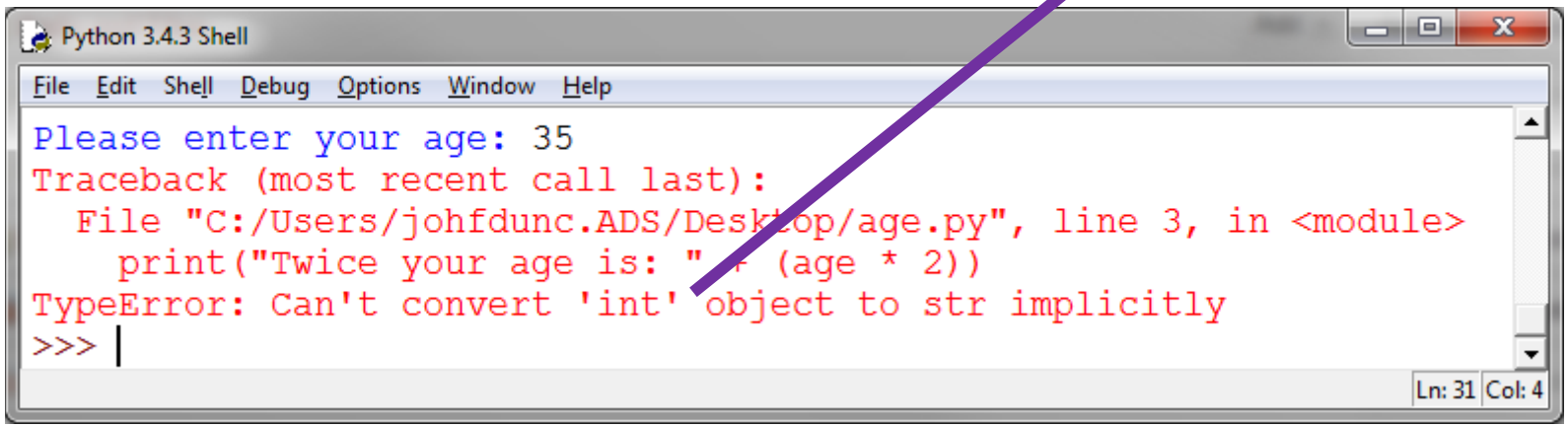


Using Data From the User

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

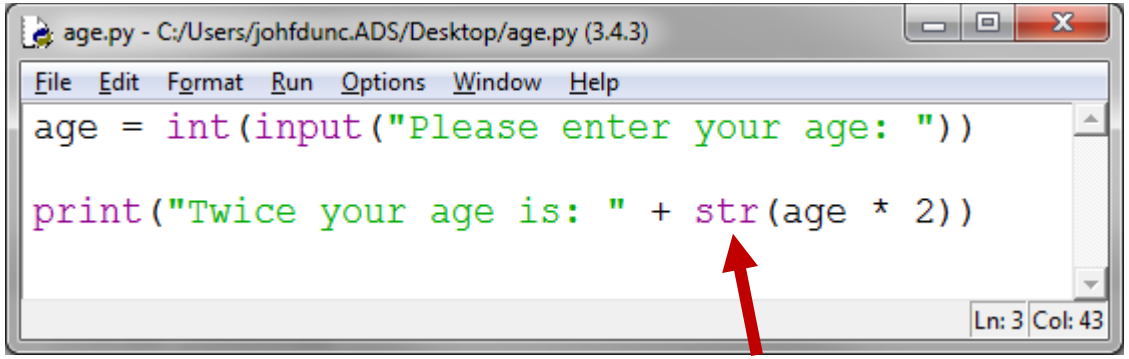
But that causes this problem:



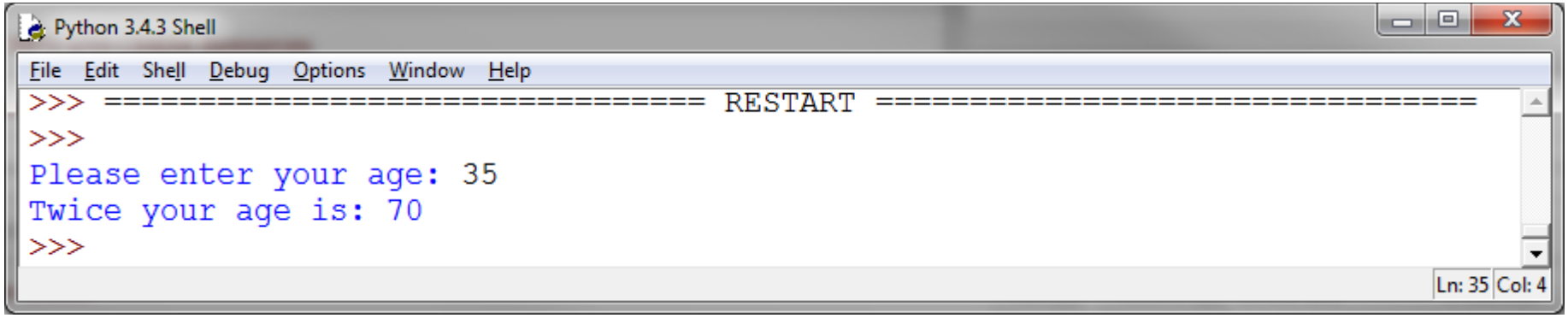


Using Data From the User

So we have to convert it BACK when we print!



Now it works!



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.

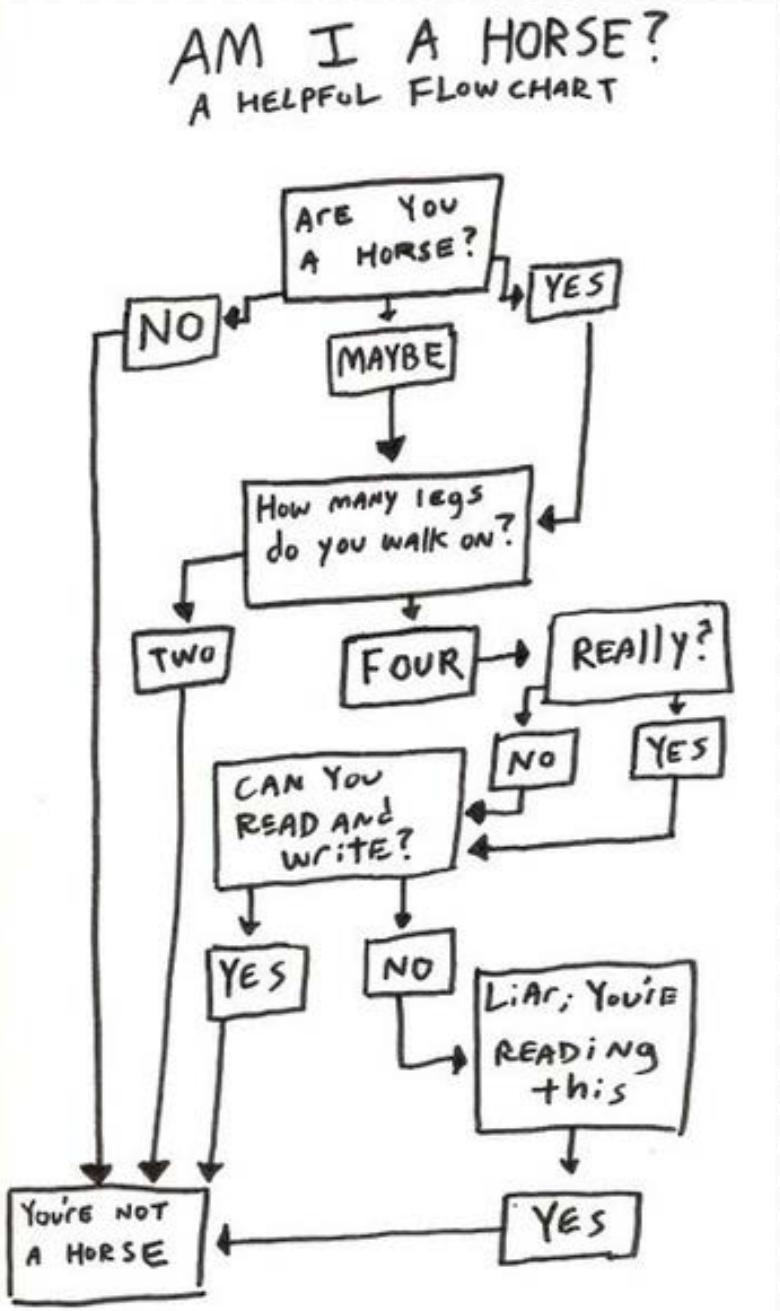
Questions?

Lecture 5

I210 – INTRODUCTION TO PROGRAMMING WITH PYTHON

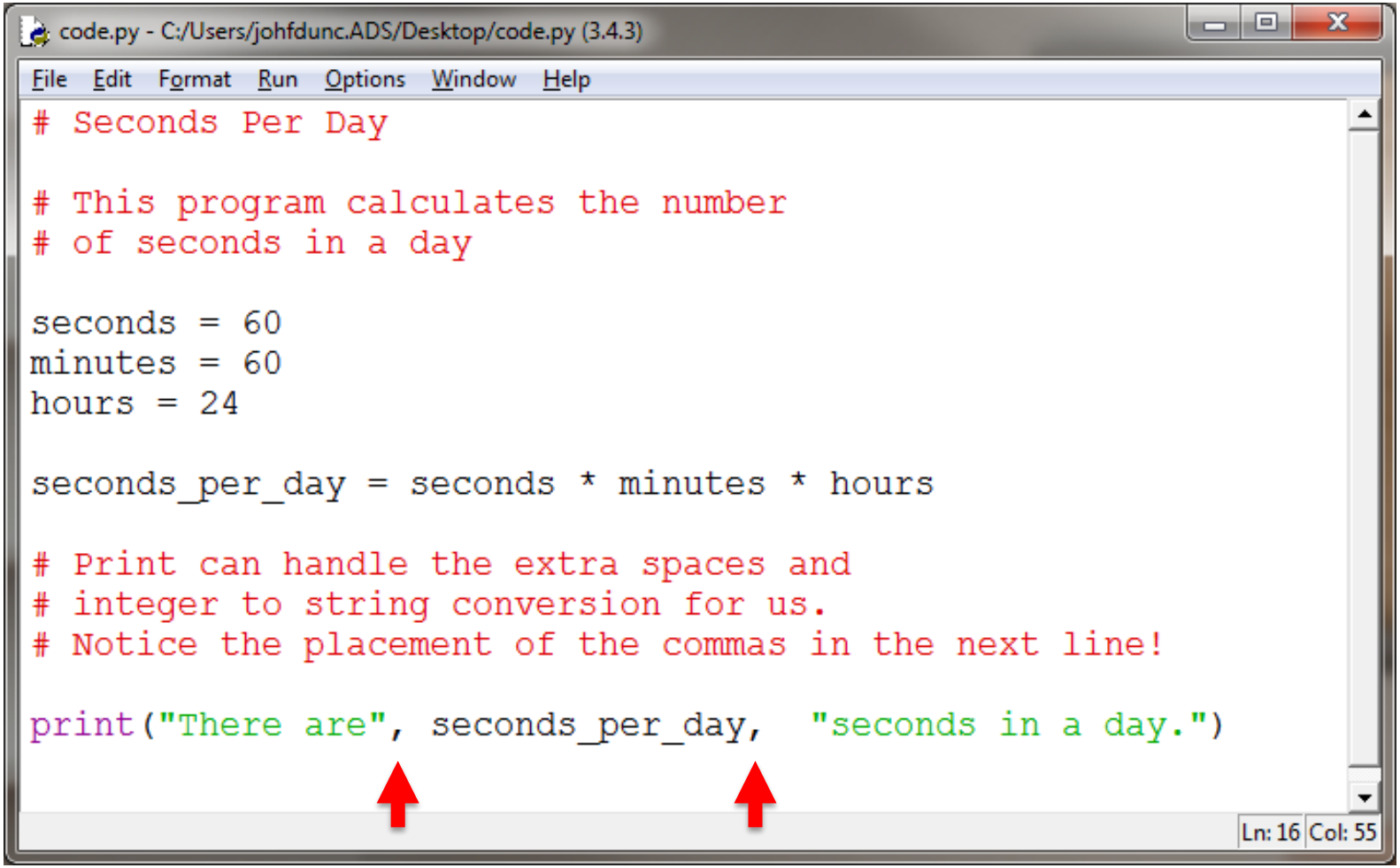
Today

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

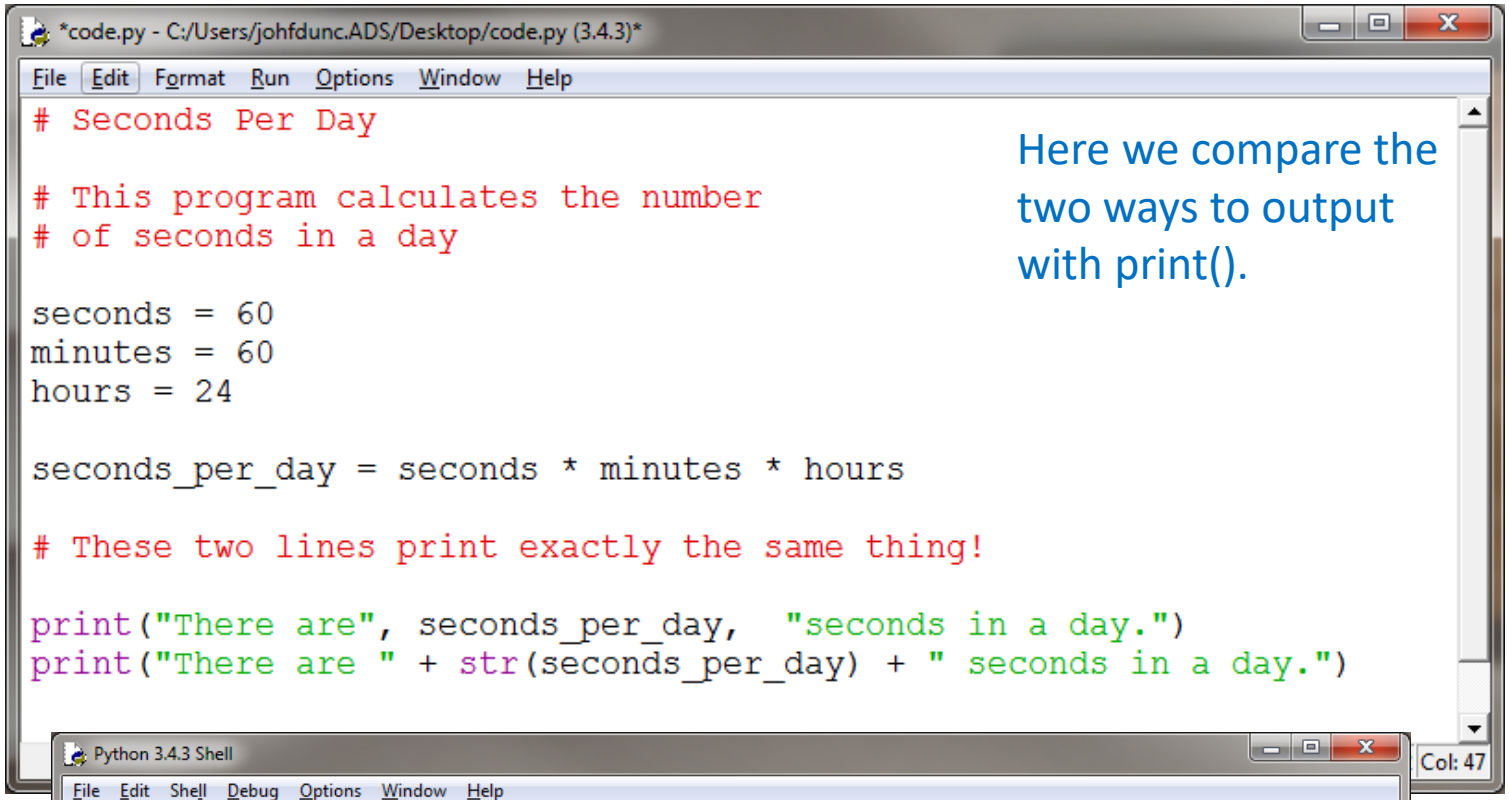


print() with commas

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



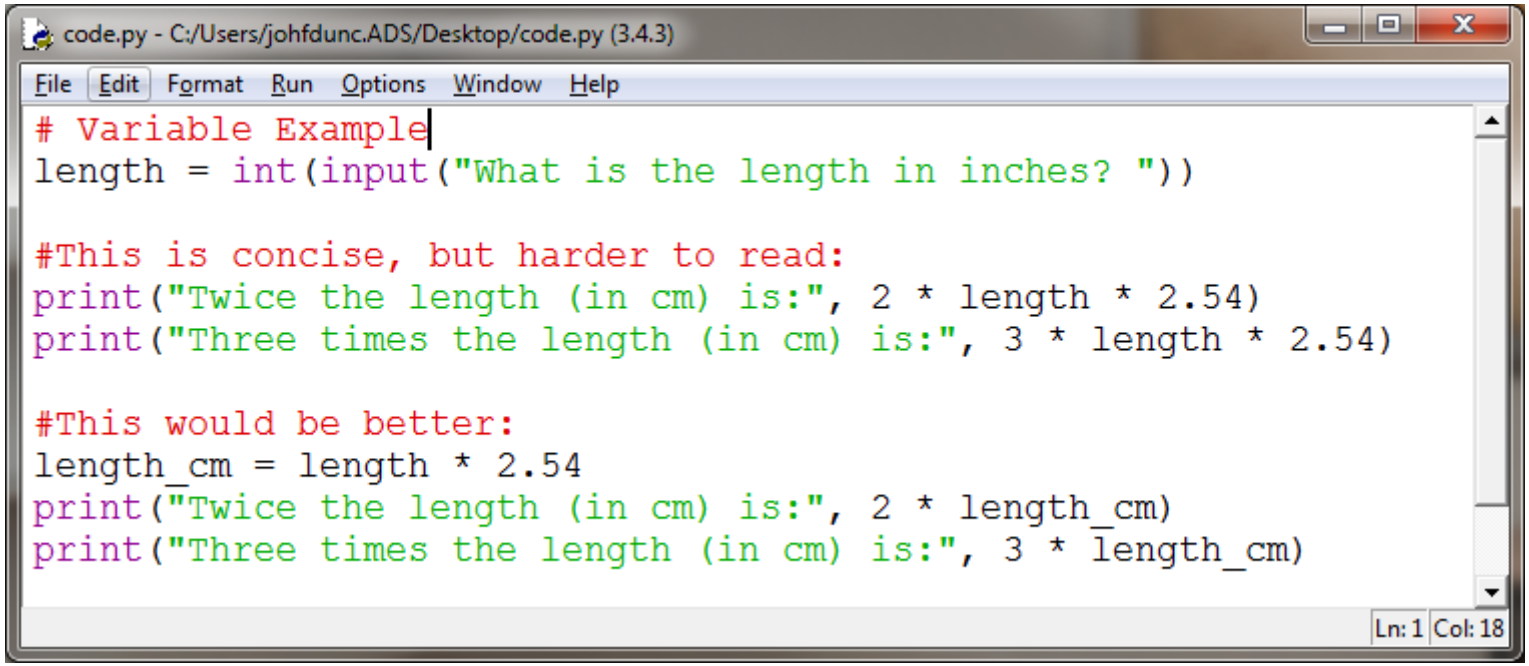
Seconds Per Day (Version 2)



Style Tip – Variable Use

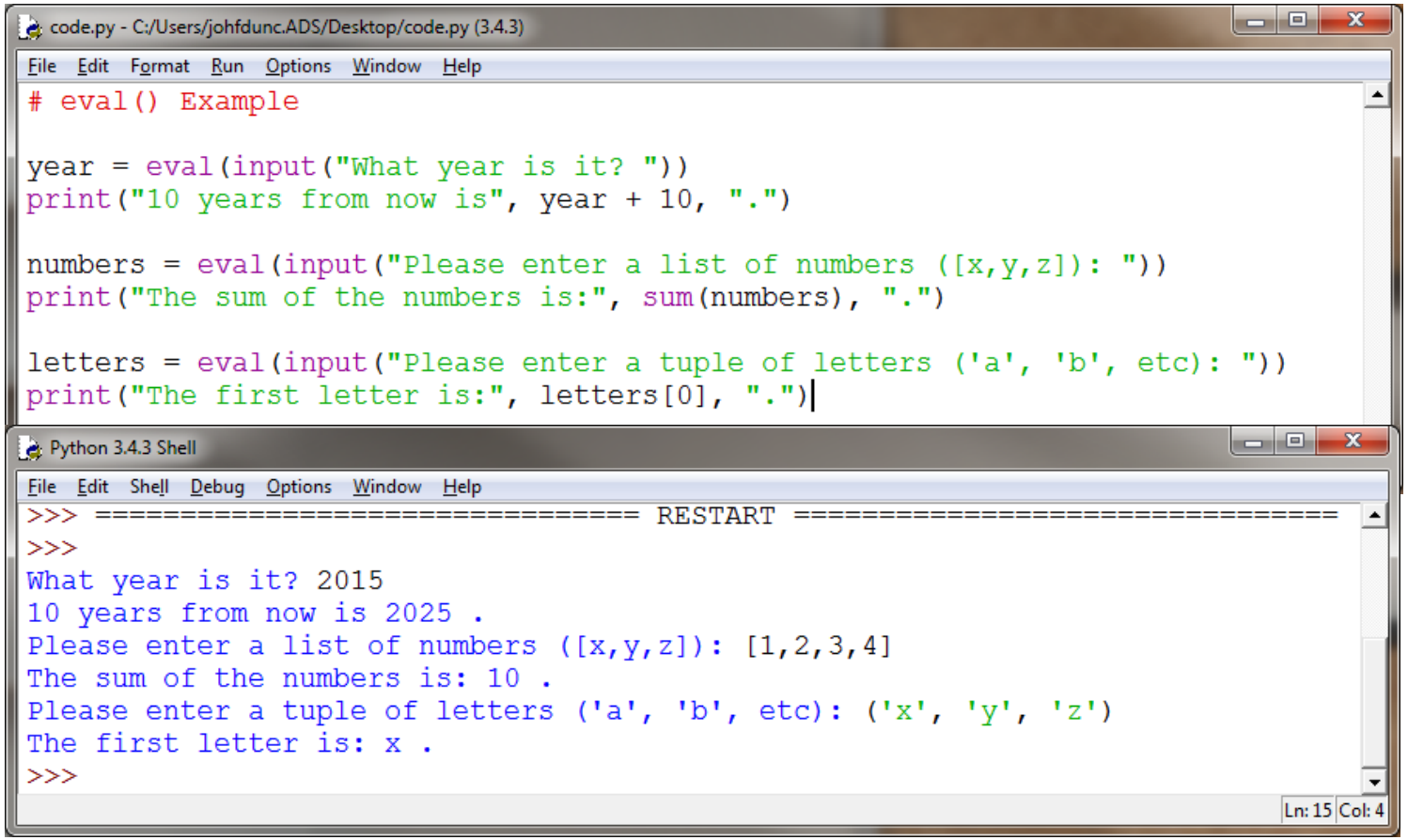
Use variables when:

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

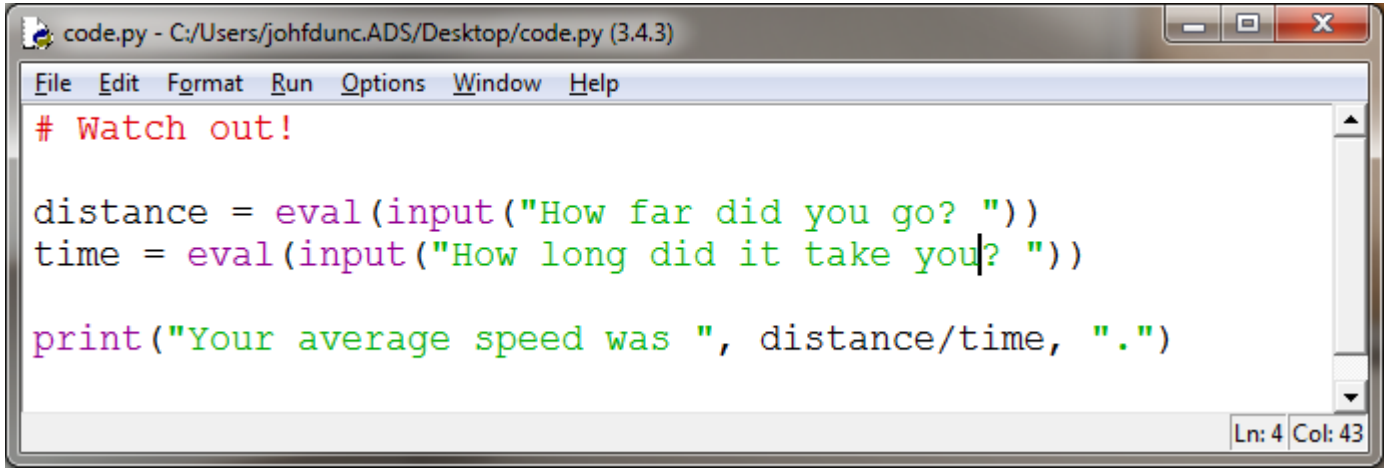


The eval() function

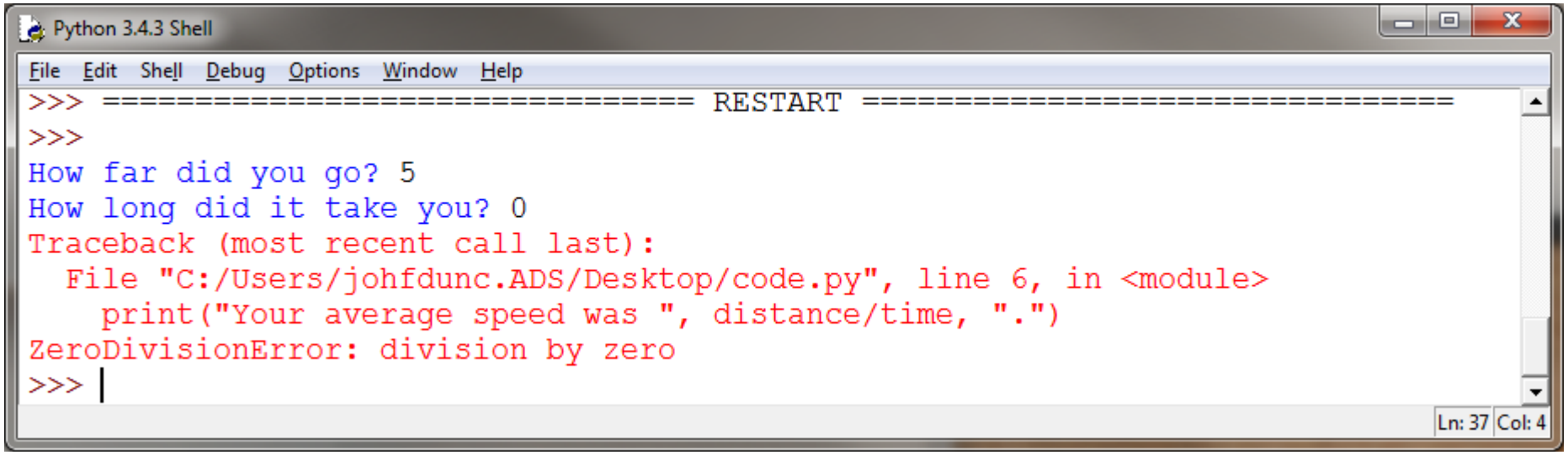
Need to get something other than a string from the user? Try the **eval()** function! Python *evaluates* the expression.



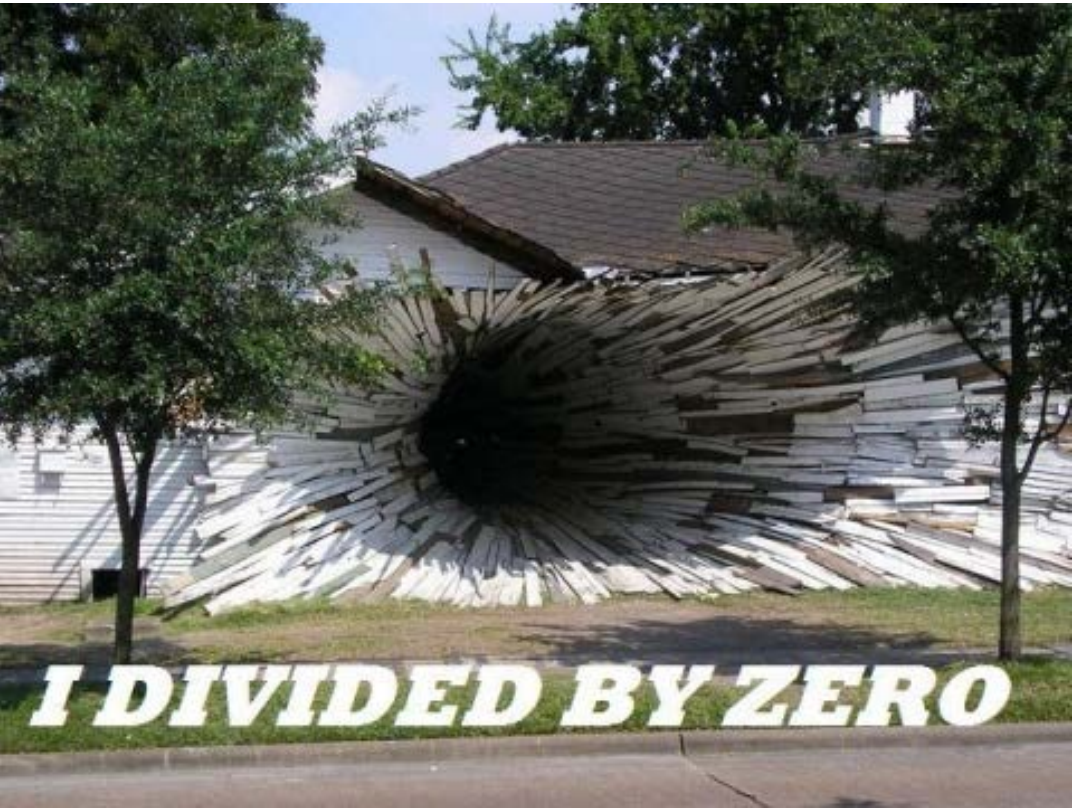
A Runtime Error Waiting to Happen



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



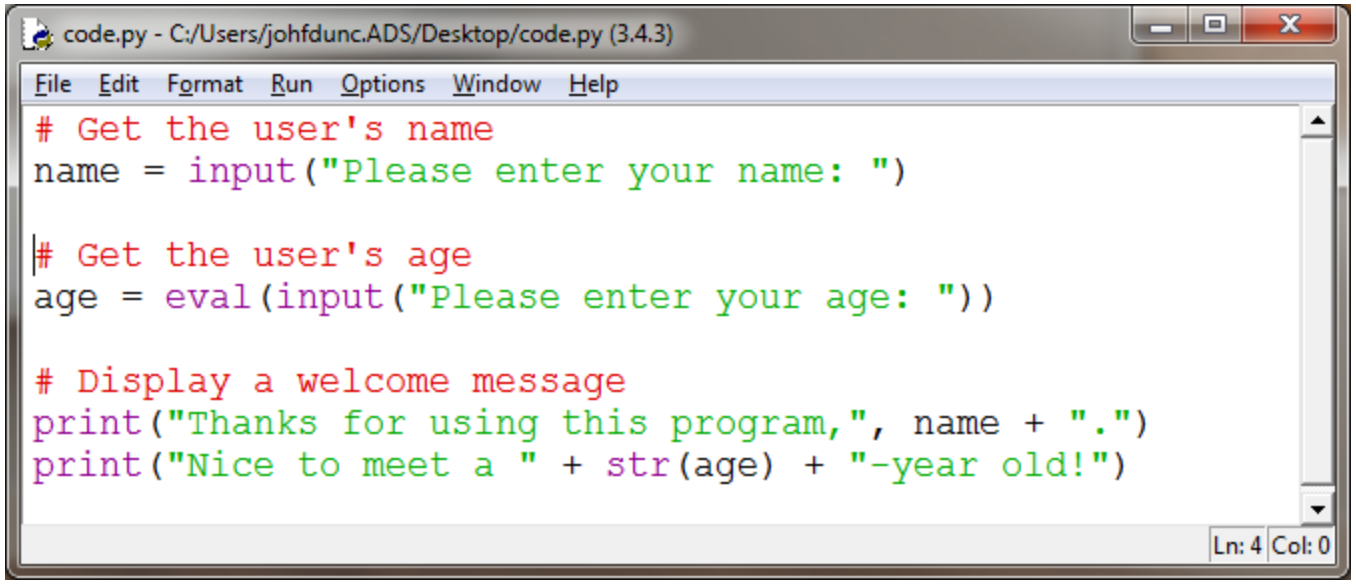
Always Watch Out for Zero Division!

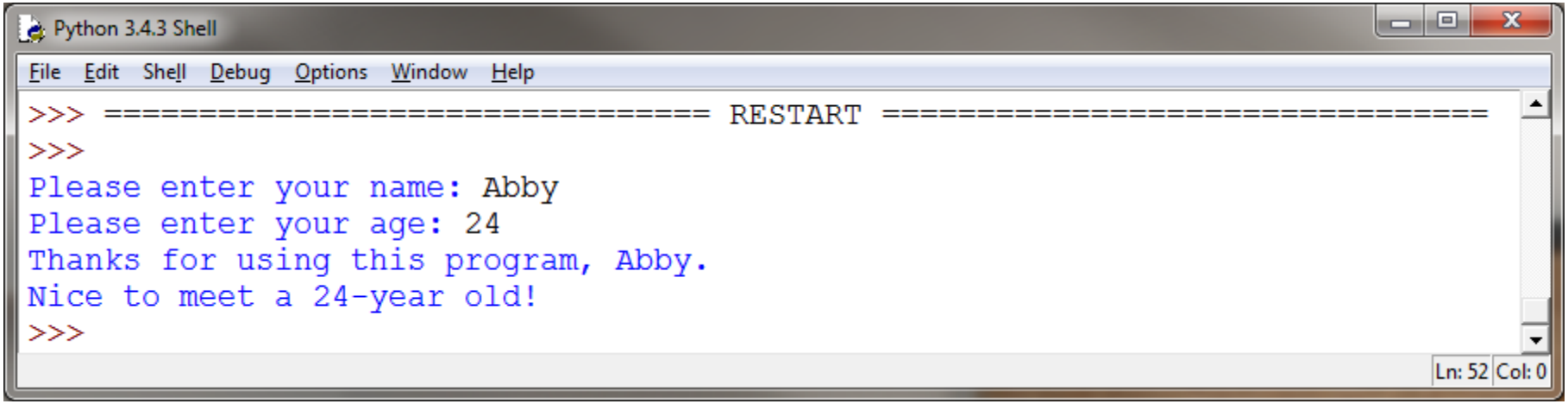


How can we guard our programs (and houses) from the terror of **division by 0**?

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

Short Greeting Program





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

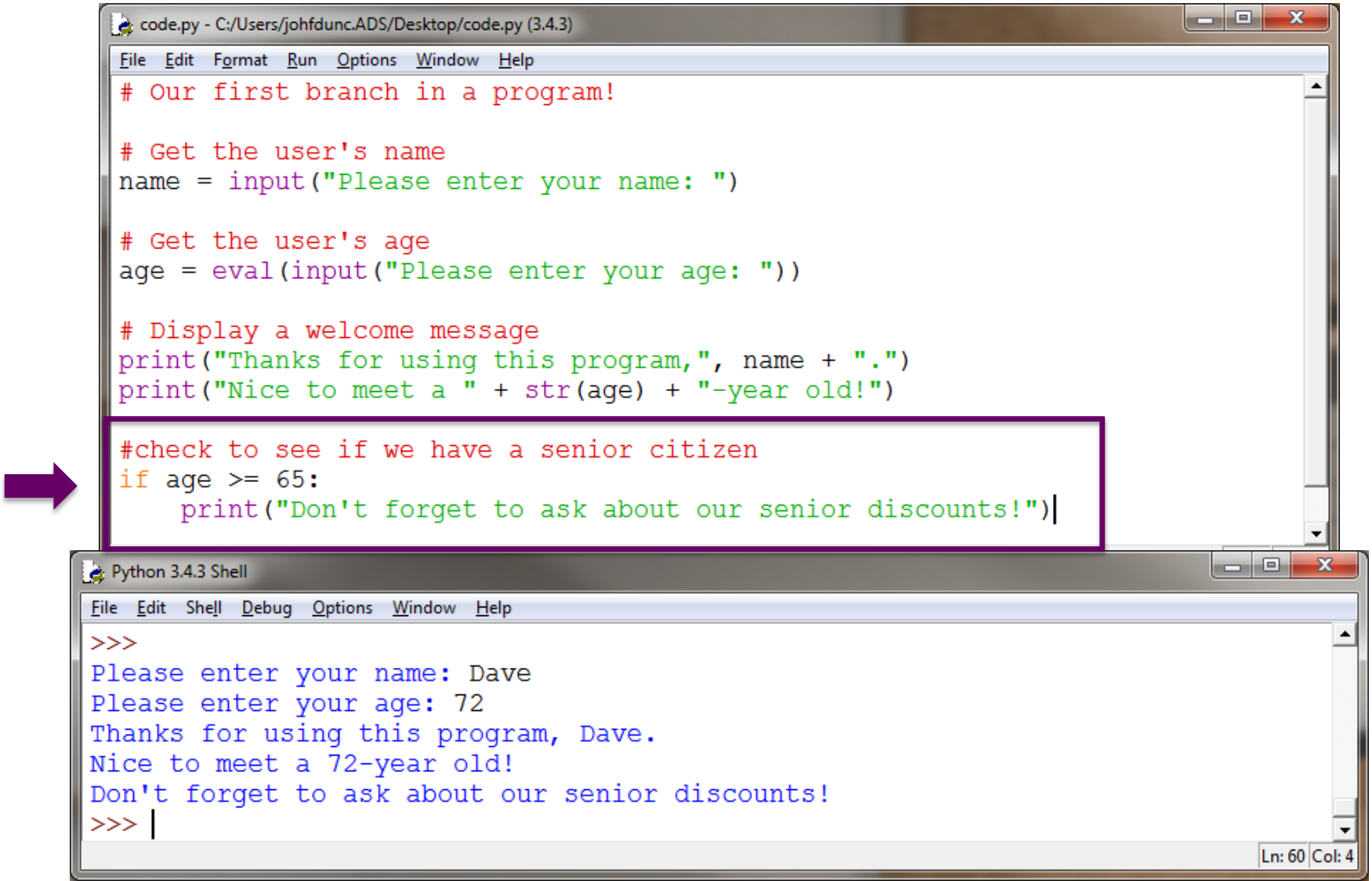
if **Boolean Condition**:

    #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



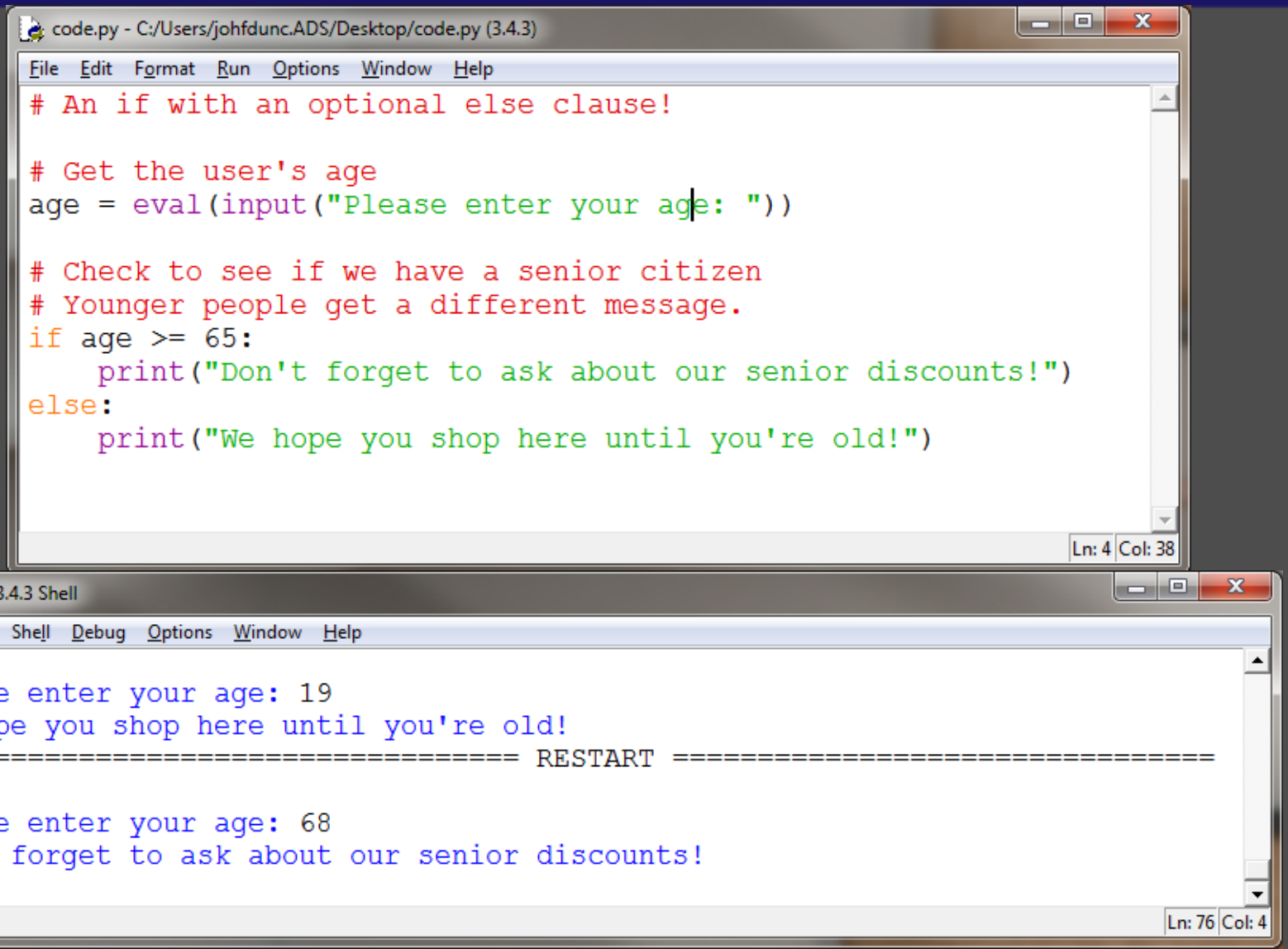
Branching – if and else

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



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*').

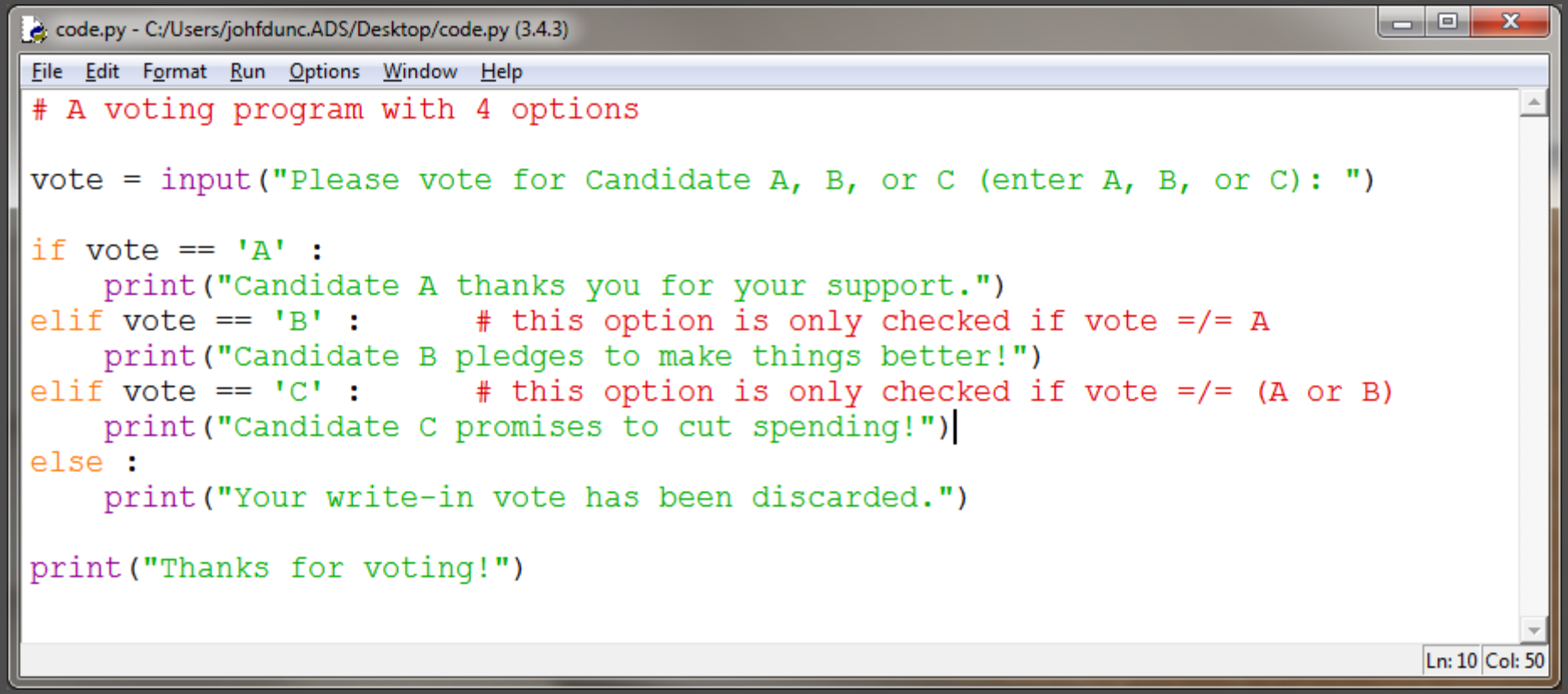
  Like else, elif is an optional "addition" to an **if** statement.

If, elif, and else statements

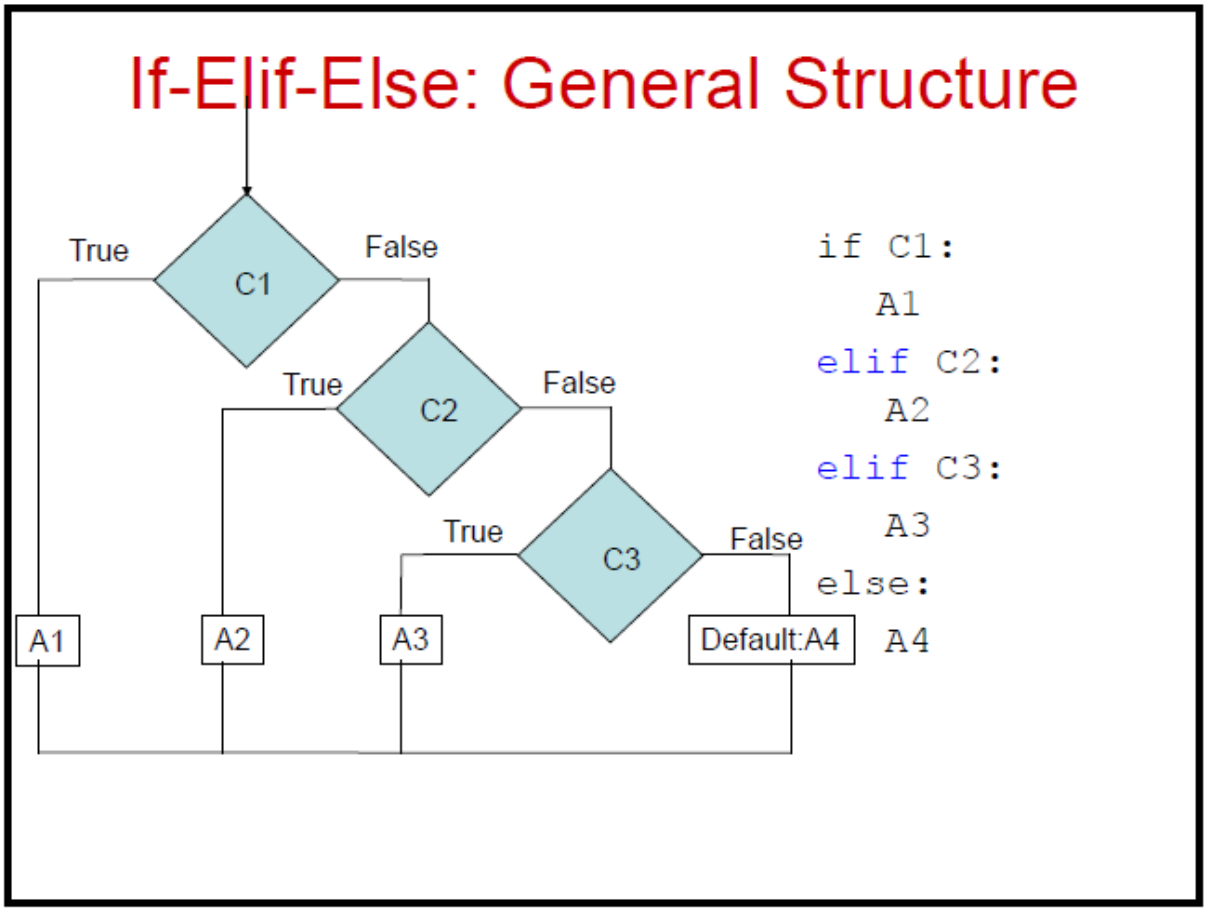
```
if  Boolean Condition:
    #Whatever we want to happen if this condition is true
elif Other Boolean Condition:
    #Whatever we want to happen if this condition is true
elif Third Boolean Condition:
    #Whatever we want to happen if this condition is true
else:
    #This will happen only if ALL 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



If / elif / else structures



The first match means we stop!!!

Questions?

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

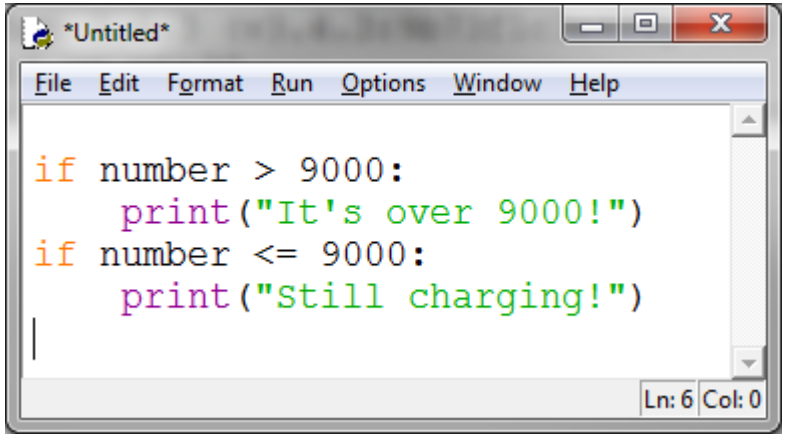
- Does the question use checkboxes that allow more than one answer to be selected? *If so, you may need to select more than one answer.*

- 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.

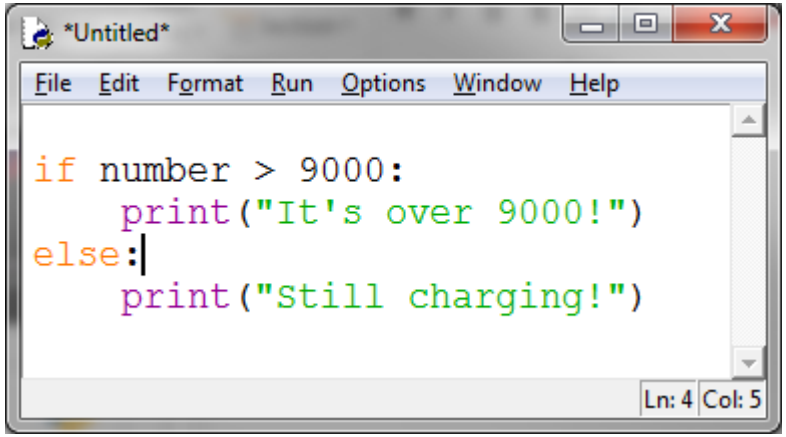
Quiz 2 – Common Mistake

- 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.



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']

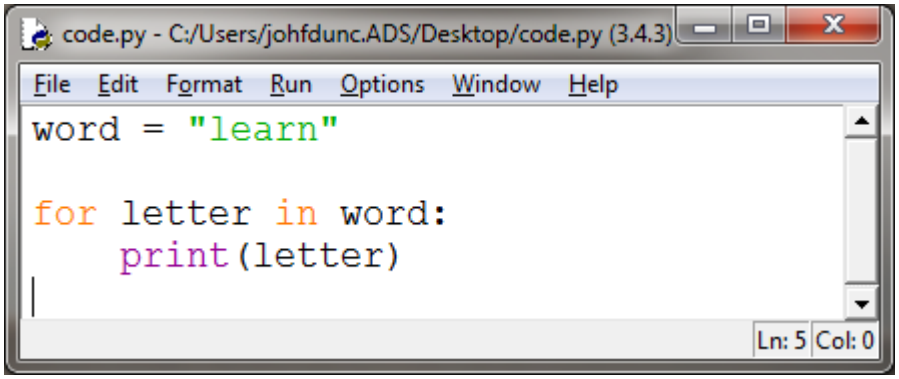
We could use a for loop like this:

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

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

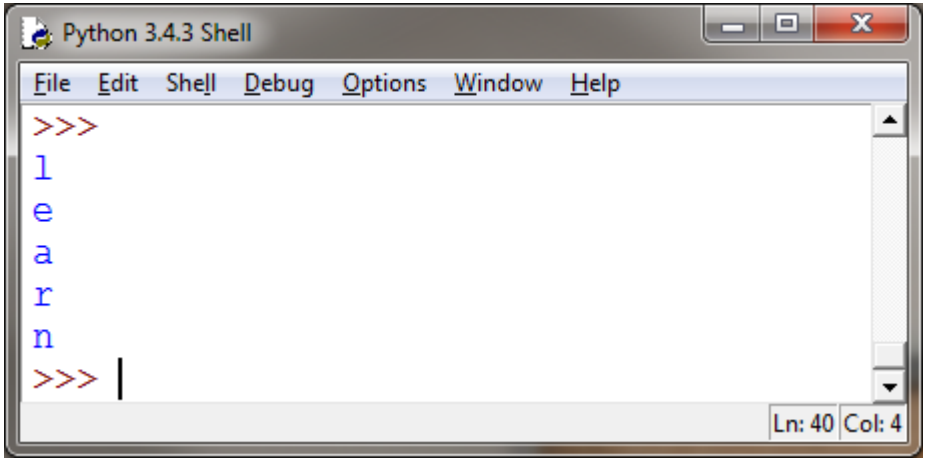
TIP: Sequence names are often plural. For loops often use a singular form of the word!

For loops, strings, in



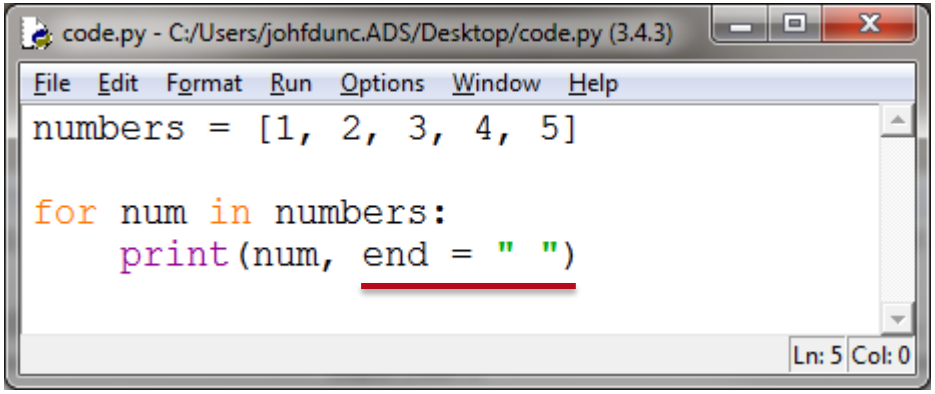
A string is also a *sequence*.

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

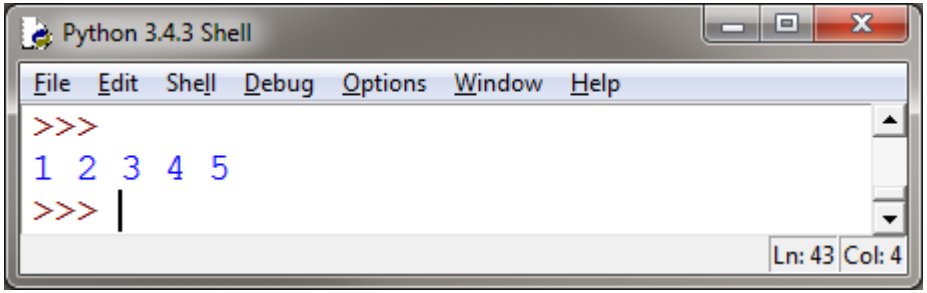


By default, if we print each element, it will be **printed on its own line.**

For loops, strings, in



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

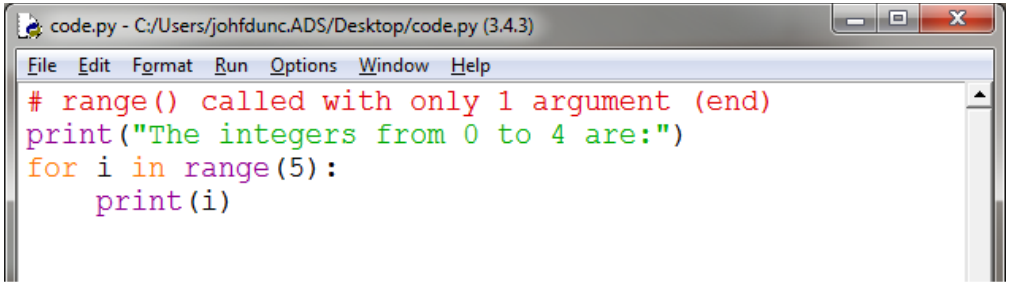


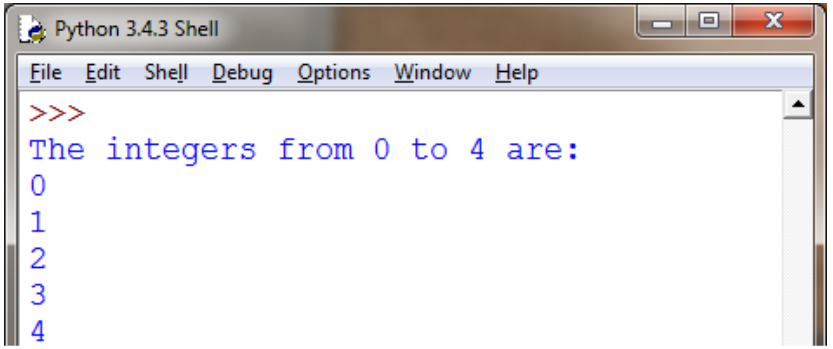
- By setting it to " " here, we cause all of the output to be printed on the same line.

This is called *suppressing the newline*.

The range() function

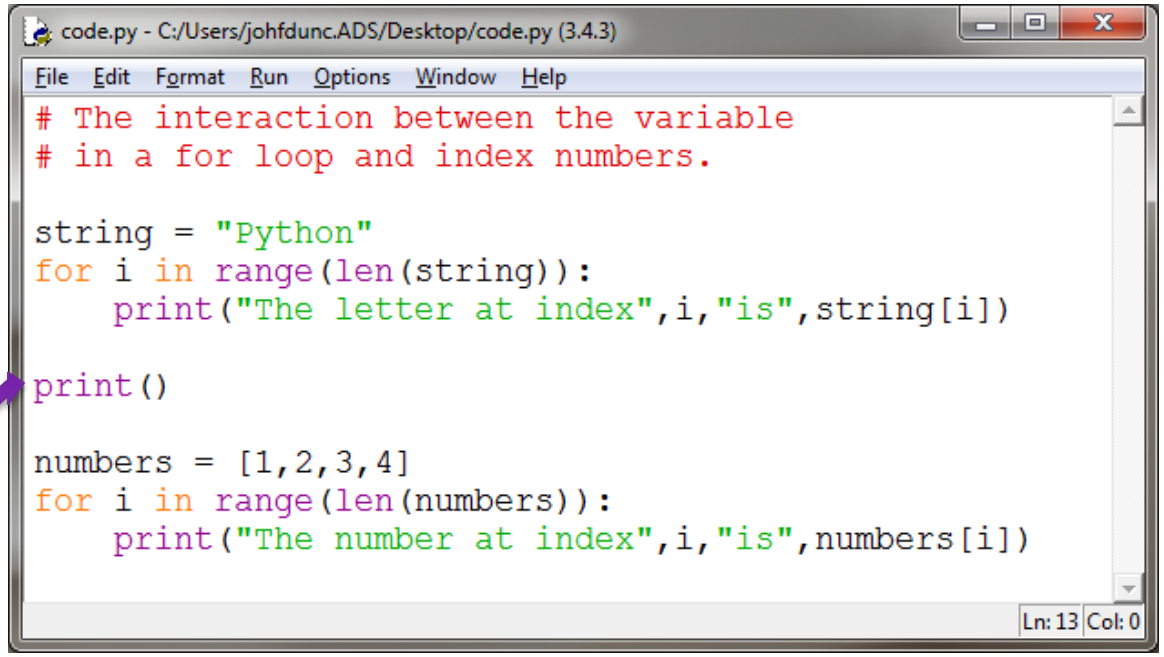
The **range()** function gives us a **sequence** of integers (a list!). *Here are three ways you can call it:*

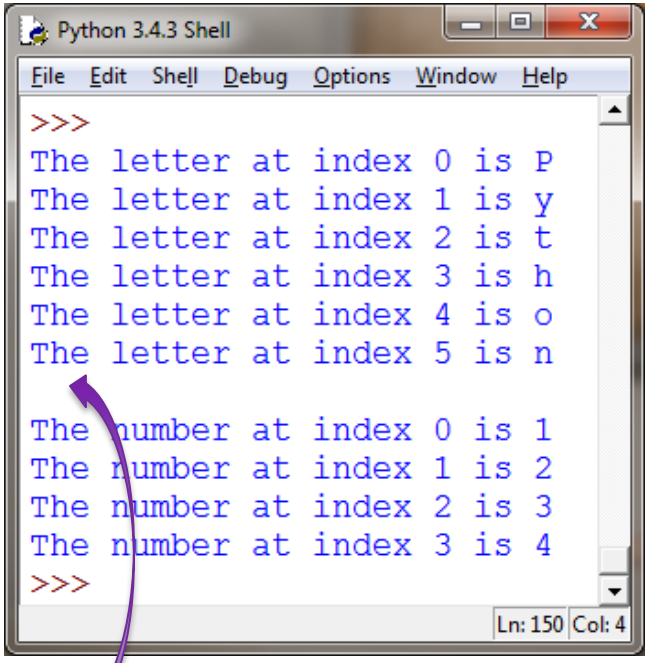




range() and len()

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

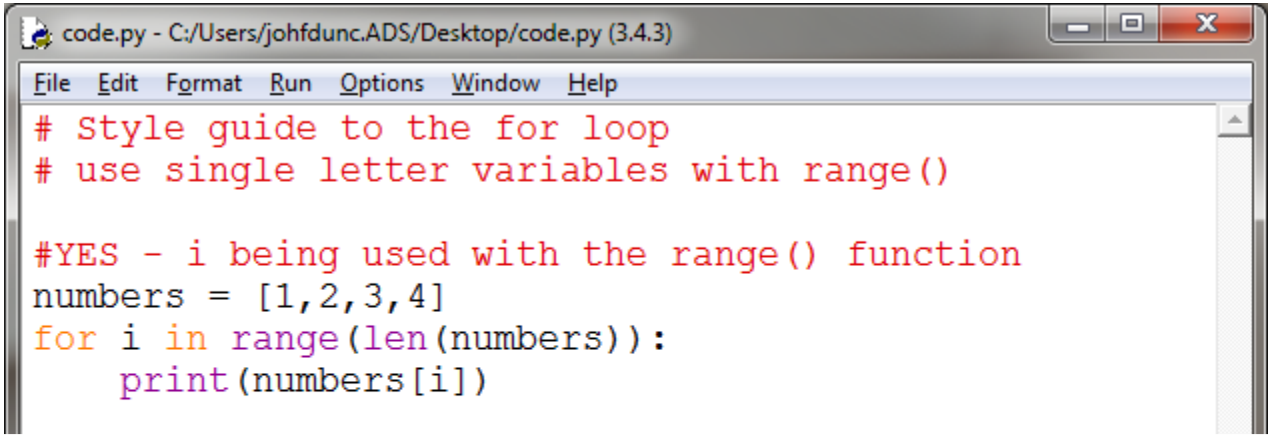




TIP: **print()** *by itself produces a blank line.*

Single Letter Variables

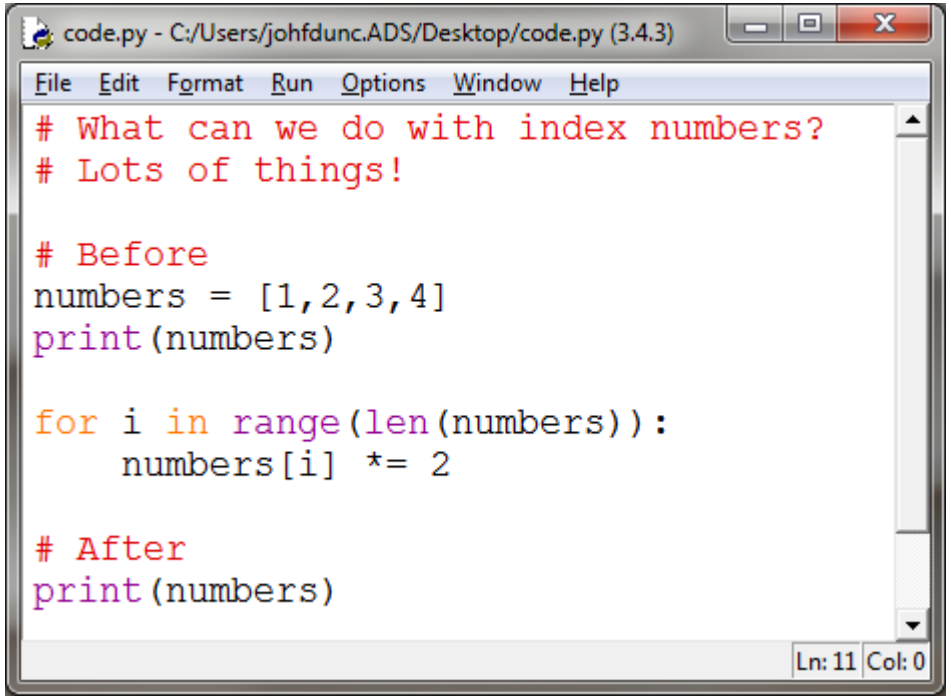
Variables with single letter names (like **i**, **j**, or **k**) are, by tradition, *only used when they represent index numbers in a loop*.

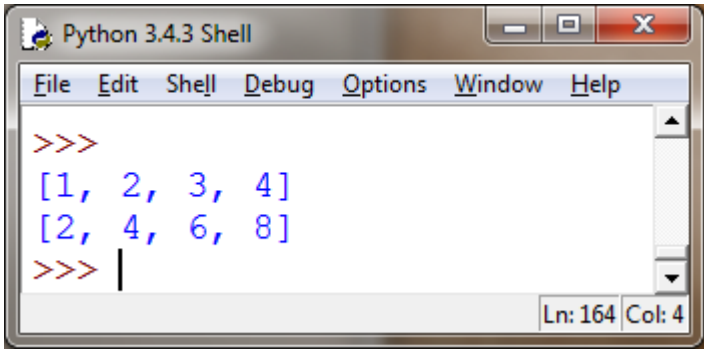


This is confusing because an

Index Numbers and For Loops

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

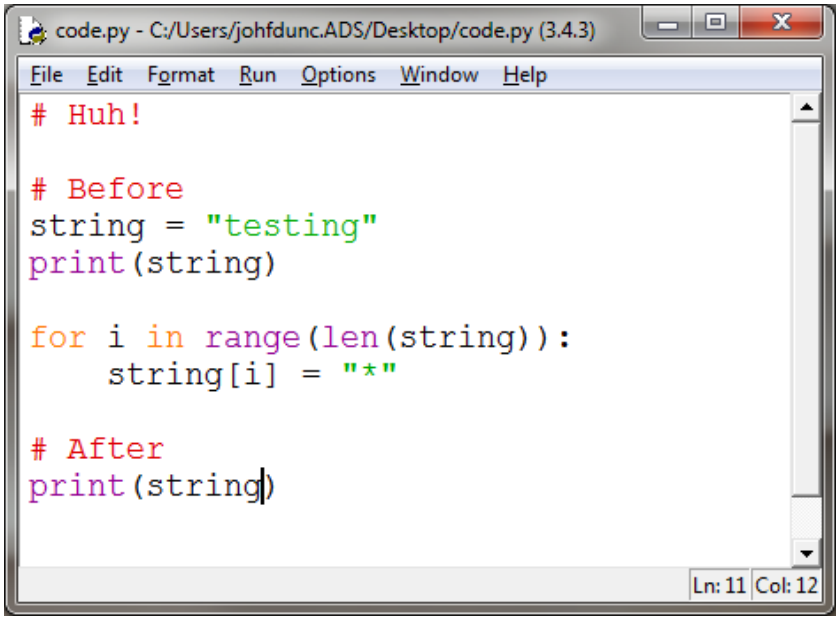


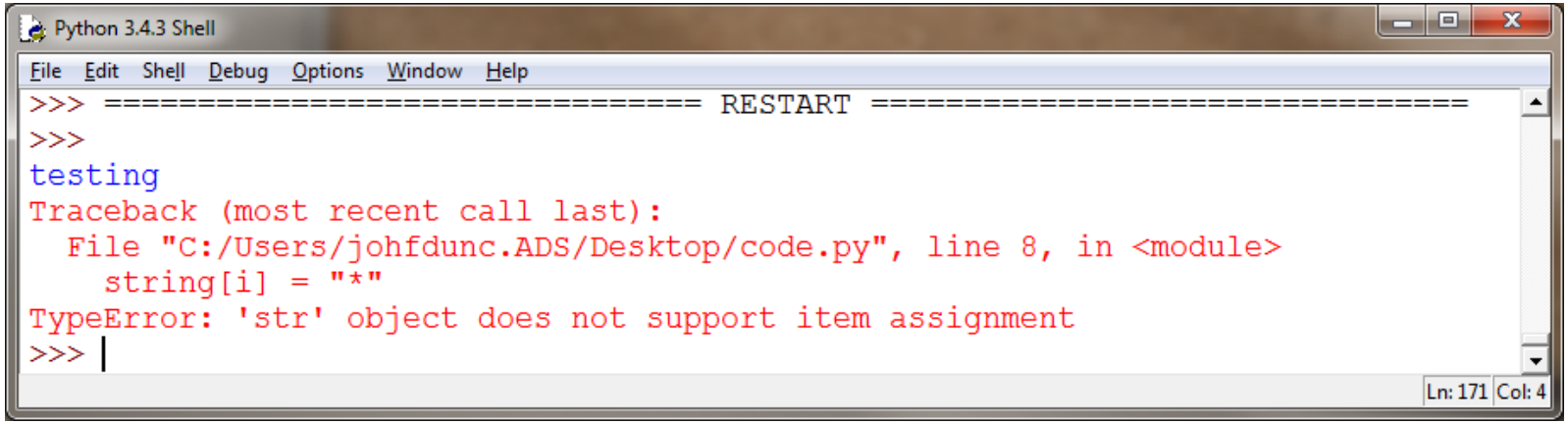


Index Numbers and For Loops

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

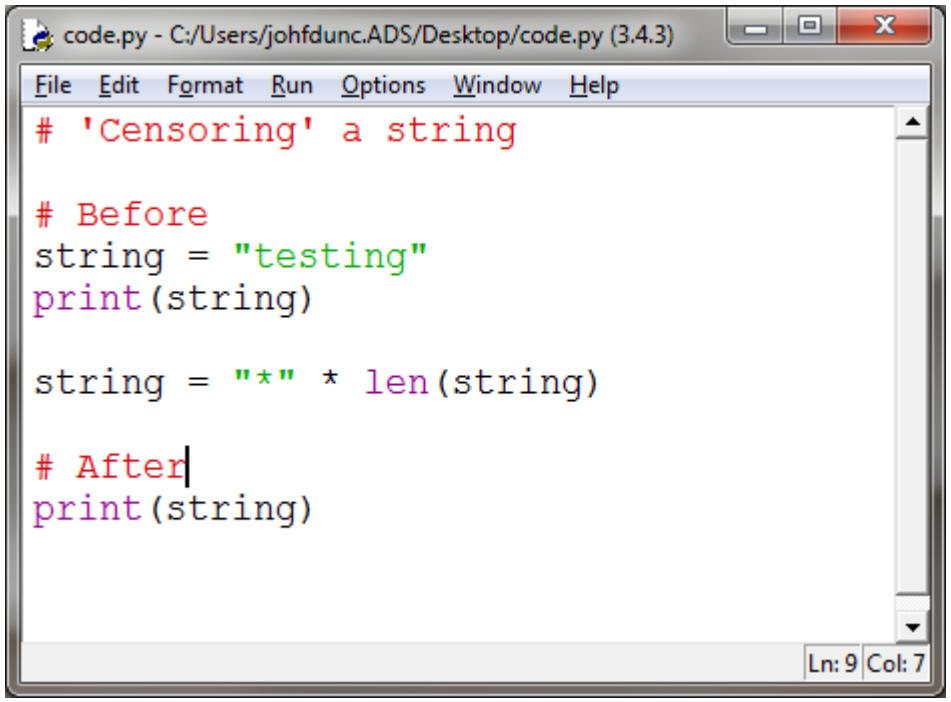
Remember, *strings and tuples are immutable!*

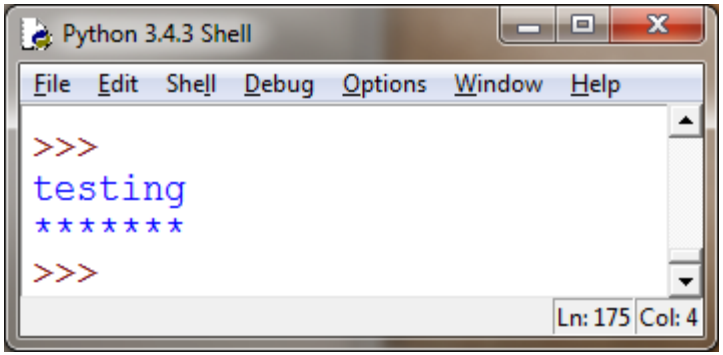




Index Numbers and For Loops

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





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!

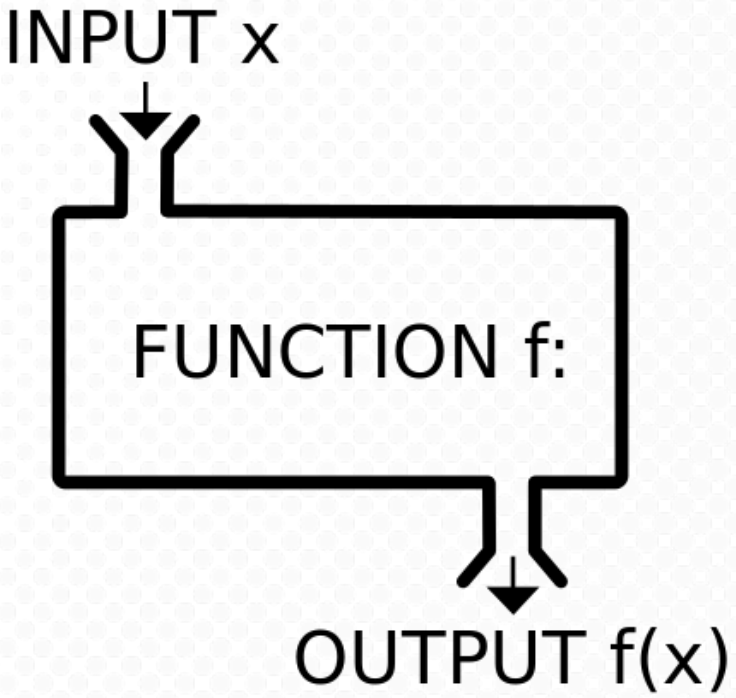
Questions?

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()
input()
len()
range()

int()
str()
float()
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.

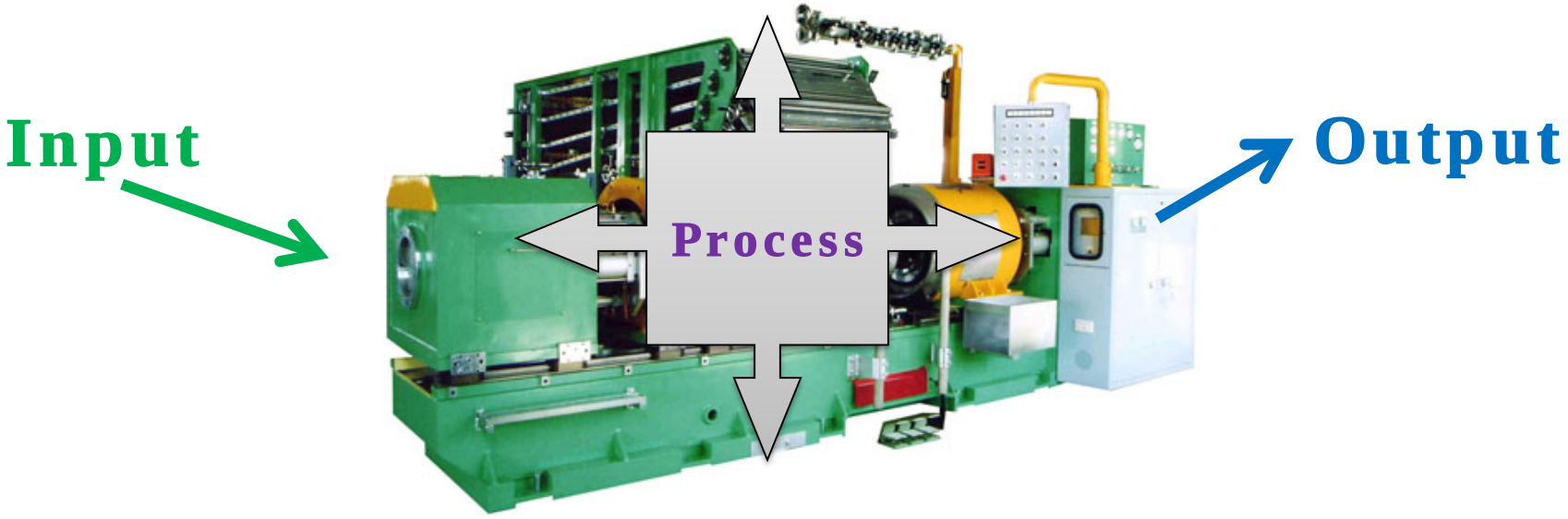
Legibility

- 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.

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

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

#main section
print(odd(6))

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

Functions

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

def odd_diff(num1, num2):
    difference = num1 - num2
    return odd(difference)

#main
user_num1 = eval(input("Please enter an int: "))
user_num2 = eval(input("Please enter an int: "))

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

odd_diff()
has two arguments!

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

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)

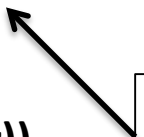
These two programs do exactly the same thing. Is one better?

```
# 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**.

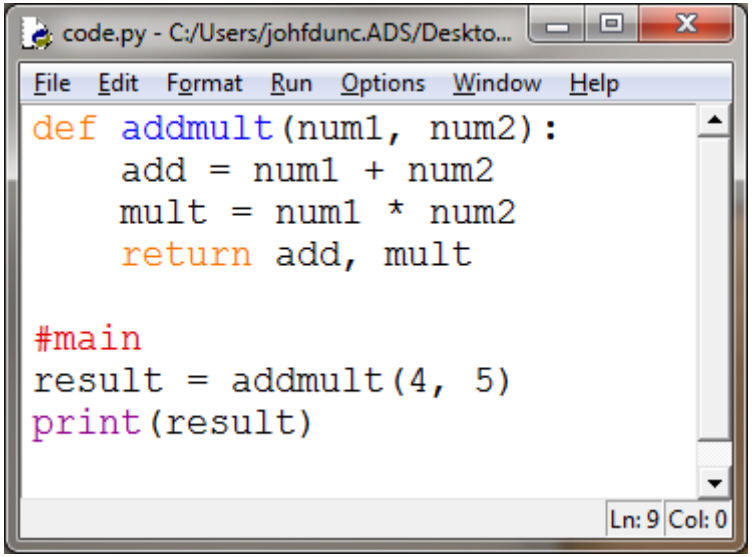
Return Values

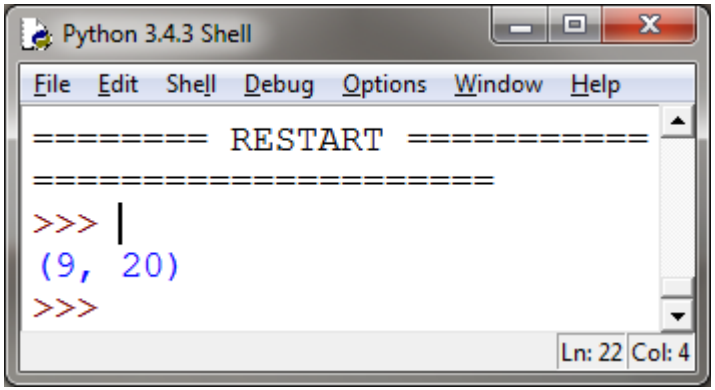
You don't have to *catch* the return value from your function (assign it to a variable), but you often want to.

- 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**:





- 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?