PROFESSOR: Good morning.

AUDIENCE: Good morning.

PROFESSOR: Thank you. OK. So we're about to launch into learning some basic elements of Python today. The elements I'm going to talk about are common to every programming language that I know, at least in concept, and of course slightly different in detail. But as I said last time, everything you're going to learn about Python should be readily transferable.

I'll be using, for all of the examples I present, something called an integrated development environment, and in particular, one that's built for Python called IDLE. Usually we talk about these things as IDEs.

I'm told, I don't know if it's true, that the IDE for Python is called IDLE after Eric Idle of Monty Python, which was also, I'm told, the inspiration for the name of the programming language.

So what is an integrated programming environment? In this case, it includes a specialized text editor that provides highlighting, auto-completion, smart indent, and you'll see shortly why all that's very important, and a few other amenities that make it easier to use this to type Python than typing it into a generic text editor.

It includes something called a shell, which is the environment that actually interprets the Python code. And the nice thing about it is it includes syntax highlighting. So the shell gives you some information about the syntax, as does the text editor of course.

And finally, it includes an integrated debugger. This could be useful in the unlikely event that your programs have errors when you write them. Though truth be told,

I've been programming in Python for years and I don't know that I've ever used the debugger. It's not that I don't make mistakes, it's just that I'm kind of a Luddite, and I typically use print statements for debugging.

And in fact, almost every programmer that I know, when push comes to shove, ends up using print statements. But the debugger is there, should you care to take a try.

All right. So you'll see on the screen here an IDLE shell. In the shell, we can type things. What are we going to type? Well the first thing to understand is that at the core of Python, and probably the most important thing to understand, are something called objects.

Everything in Python is an object. So every kind of entity that you can create in Python is an object, and in fact, Python code itself is an object. You'll remember, we talked about stored program computers last time, and the concept that a program is data, just like a number is data.

Each object has a type that tells us the kind of object it is, and in particular, what we can do with it. And then there's a built-in function, called type, that can be used to find out the type of an object.

As we'll see, there are two fundamental kinds of types. Scalar and non-scalar. We'll start with talking about scalar types. And the key thing to think about there is that they are indivisible. Think of them as the atoms of the programming language.

Now, I know that some of you have studied physics and know that atoms are in principle divisible, but of course, only at great expense and with serious consequences. And we've seen the same thing here. You can, if you're desperate, chop up these scalar types, but it almost always leads to something bad.

All right. Well let's look at some. Well, the first one you'll see is used to represent integers, and that's called int. For every type, or every built-in type, there's the notion of a literal, which is how we type it.

So for example, we can type 3, and that will now tell us it is the value 3. You'll note

it's typed it in blue. And I can ask what the type of 3 is.

So you'll notice as I type things into the shell, it's using colors to give me a hint. So it's this fuchsia color for the word type, telling me that's a built-in function. And now if I ask it, it will tell me that the type of the literal 3 is "int". So it's an integer.

And I can use other sorts of things. All right. There's also a type float. So those correspond to the real numbers. So I can do something like that.

And we'll talk about this in a second, but you'll notice if I do type of 3.2, it tells me it's a float. And for that matter, I can do type of 3.0, and it will tell me it's a float. So there's a difference between 3 and 3.0. One is an int, and one is a float.

Now you'll notice something kind of weird here. When the interpreter printed back the value of the literal 3.2, it gave me 3.2 and a bunch of zeroes, and then this funny 2 standing at the end. In a few lectures, I'll explain why it does this, but for now, you should just take this as a warning that floats are not the same thing as real numbers.

You learned about reals, presumably in middle school or high school. Floats are a computer scientist's approximation to reals, but they're not quite the same.

The good news is almost all the time, you can pretend that a floating point number is a real, but as we'll see in a few lectures, every once in a while it can really sit up and bite you, if you believe that. But for now, we'll just pretend that they're reals.

There's Booleans, a nice scalar type of which there are only two values. One of them is true, and what do you think the other Boolean value is?

AUDIENCE: False?

PROFESSOR: Thank you. So somebody said false. I have no idea who, but whoever it is, there's probably some candy to be had. Oh, I managed to find the one place in the room where there was an empty. I'm hoping that people will now scramble and fight for it, like a foul ball at a baseball game. No, people are too polite thus far.

All right. So we have true and false as the type Booleans. And then we can do operations on them. So for example, true and false is false, as you might guess.

Finally, there's this funny value, none, which doesn't print anything when I type it. And if I look at the type of none, we'll see it's the none type. Not very interesting.

Fundamentally, as we'll see, that gets used when you want to put in something temporary. When you don't yet know what its value is going to be, you know it's going to eventually have one, so maybe you start out calling it none. And then you can check, and we'll see how we might do that.

So those are the fundamental scalar types, the indivisible ones. Interestingly enough, Python does not have what is a common scalar type in every other language called char, short for character. Instead, what it has is strings that can be used to represent strings of characters.

So for example, I can write the string "a", and if I ask for the type of it, it tells me it's an str, short for string. Happens to be a string of length 1, which we might usually think of as a character, but there is no type char. So it's not a problem. We just have to remember it.

Literals of type string can be written with single quotes or with double quotes. There's no difference. Just convenient that you can do it either way, and we can build strings of things. It's worth noting that the type of, say, the string 123 is str, whereas the type of 123 without the quotes is int. So we have to be a little bit careful sometimes as to whether we're dealing with strings or ints when we look at these literals.

You can only get so far with literals, things you can type. So of course, Python has in it something called an expression. Again, this shouldn't surprise anybody. And an expression is a sequence of operands and operators. The operands are objects.

So for example, we can write the expression 3 plus 2. And when we type an expression into IDLE, it automatically evaluates it and prints the value of the expression. In this case, of course it's 5.

One thing to be a little careful about is if I type the expression 3/2, slash is the divide operator, I get 1. Whereas if I type the expression 3.0 divided by 2.0, I get 1.5. So dividing two integers in Python 2.x gives you essentially a floor operator. In 3.0, by the way, integer division is not allowed. It always converts it to floats and does a floating point division.

But for many of you this will be something that will trip you up as a bug. If you want to get real division, write floating point numbers. Otherwise, unpleasant things may happen.

Some other interesting things I can type, just as I could type 3 plus 2, I can type a plus b. What do you think I'll get there? It does concatenation.

So what we see here is that the operator plus is overloaded. So overloaded operators have a meaning that depends upon the type of the operands. And of course, we've already seen that with the slash operator, which means one thing for ints and another things for floats. And of course, we see the same thing with plus.

What do you think will happen here? 3 blank 3? Any guesses? I get a syntax error. Remember, we talked about that on Tuesday. It's not a valid Python expression, so we get an error.

How about this one? That is syntactically valid. It's got operand, operator, operand. What do you think it will do when I hit Return here? Somebody?

AUDIENCE: A static semantics error?

PROFESSOR: Pardon?

AUDIENCE: A static semantics error?

PROFESSOR: A static semantics error. And because of these-- Wait, I can't see who said that. Raise your hand? Oh, come on. All the way back there? All right. I have the most chance of carrying with one of these. I'm going to lie. Those of you who are watching OpenCourseWare, it was a perfect throw.

OK. So indeed, we get a static semantic error of a particular kind, called the type error, saying you cannot concatenate an str and an int. Type errors are actually good things. The language does type checking in order to reduce the probability that a programmer will write a program with a meaning that will surprise its author.

So it looks at it and says, somebody might have a weird guess what this means, but just to be safe, we're going to disallow it rather than-- it could, of course, make up some funny meaning if it wanted to. But it doesn't. And I think you'll find type checking saves you from a lot of careless programming errors as you go on.

All right, let's continue. Let's look at some other things. I can write this. Because that's just two strings, and it just concatenates them, the string a and the string 3. Or interestingly, I can do this.

So now what we're seeing is that you can take any type name, use it as a conversion function to attempt to convert one type to another. So this has now converted the int 3 to the str "3".

Similarly, I can do something like this. And here, it's converted the str "3" to the int 3. On the other hand, I could do this. And it will tell me it's a static semantic error. It can't convert 0.0 into an int. Similarly, it can't convert 2.1.

Or can it? So now I've given it the float 2.1, and I've tried to convert it to int. Not the string 2.1, but the float. And it succeeds. And it succeeded by essentially truncating it.

Is this a good thing or a bad thing? To me, it's kind of a bad thing. If I've typed something like that or I've evaluated some expression that happened to work that way, more likely than not, I'm confused. And I would probably have preferred to get a type error, rather than it deciding how to do it.

It's one of the things I don't like about Python. It's too generous. It lets me get away with stuff it shouldn't let me get away with. Other languages, for example Java, are much stricter. This is a design decision and it is the way it is, and we have to live

with it.

AUDIENCE: Professor?

Yes?

AUDIENCE: Is that the same reason that 3 divided by 2 turned into 1 up top?

PROFESSOR: Yeah. Exactly. If it's the same reason that that happens, this will never go that far. [UNINTELLIGIBLE]. Yeah, exactly.

It's the same reason. The question was, is it the same reason that 3 divided by 2 doesn't give you the answer you would get with floating point. And it's because Python has tried to help you. Again, Python 3.0 is a little stricter about these things. We'll talk much more about this during the term.

This is close to the last time you'll see me typing things directly into IDLE. For the most part, as you write programs, you'll use the text editor to produce them and then go to the shell to run them.

But you want to-- obviously, if I had a 100 line program, I wouldn't want to sit here and retype it every time I needed to change it. So instead, I use the editor in IDLE to produce the programs, and then I can run them. And that's what I wanted to start doing.

I should probably mention that what most people call a program, some Python programmers call a script. Think of those two things as synonyms. But you will see people use both of them. I will typically call them a program.

All right. Let's look at an example. So the first thing to say is that things look a little bit different when they're executed from a script than when you execute them directly in the interpreter. So I happen to have a script here.

If a line in a script starts with a sharp sign or a number sign, that makes it a comment. So it's not executed. So I've started here just by commenting out everything.

But now-- whoops-- what happens if I just put the number 3 here? We saw when I typed it into IDLE, it echoed it in some sense and gave me what it was. Or just to be clear, I'm going to put in the expression type of 3. I'll save it, and then I'll hit F5 to run it.

And it does nothing. Right? You saw it move. It didn't print anything.

So when you type an expression into the shell, it prints the value. But when it executes a script with an expression, it evaluates the expression but does not display it on the screen. Well, so what do we do about that?

There is something called a print command. So I can do this, Print type of 3, and now if I run it, it will actually appear. So whenever you want to get something to appear, you have to use the Print command. Not a very complicated concept.

A program, or a script, is a sequence of commands. Each one tells the interpreter to do something. So a command is Print, for example.

OK. So that's there. That's kind of boring. I'll get rid of that.

The next command is a really interesting one. It's an assignment statement. A key concept in almost every programming language is that of a variable. Different languages have different notions of what a variable means.

In Python, a variable is simply a name for an object. And what an assignment statement does in Python, is it binds the name to an object. So the assignment stetement you see here binds the name x to the object 3.

The next statement rebinds the name x to the value of the expression x times x. So it takes the old value of x, evaluates the expression, and then binds the name x to the new value. So at the end of the second statement, x will be bound to 9.

By the way, these are really stupid comments I've written here. I put them in just to show you what these statements are doing. For goodness sake, when you write comments in your programs, assume that the reader can read Python, and you

don't have to explain the programming language in your comments.

That's not to say you shouldn't write any comments. The purpose of a comment is to make the program easier to read. And so typically, comments are there to explain things. Not to explain the language or its semantics, but to explain your thinking when you wrote the program. What is the algorithm you've used? And we'll see some useful examples of comments, probably not today, but later.

All right. So let's execute this script. Sure enough, it printed 9. Just what we would have hoped.

All right. Now let's try some other things. Print lets us output things. Raw input lets us input things. Get things from the keyboard, essentially. So this statement here is making a request to whoever is using the program to enter a number.

There are two kinds of input statements in Python 2.x. There's raw input, which is the only one you will see me use, and input. Raw input, by the way, is the only one that exists in 3.0. So please, just use raw input.

The difference is, raw input always expects, interprets what the user types as a string. So it will see here, it says, y equals float of raw input. Enter a number.

So let's run it. So it's taken the argument to raw input, the string enter a number asked me to enter a number. I'll enter a number. And then it's converted it to a float.

Suppose I get rid of that. Suppose I do this. That should work. So now something has happened. It's printed both of them as 3.0. It looks like they're the same, but in fact, they're not. And this is something to beware of.

What we've seen here is when it prints a string, it does not print the quotation marks. So even though, if I were to put this in here, I'll put in two print types of y. And I'll comment this out because I'm getting kind of tired of seeing 9.

You'll note that one is a string and the other is a float. Again, I point this out because this is something that can confuse people when they're debugging programs. Because you think it's a float, when in fact it's a string. OK. Nothing deep, but these

are the things that sort of get people in trouble.

Now the kinds of programs we've been looking at so far are what are called straight line programs. What distinguishes a straight line program is it's a sequence of commands you execute one after another. You execute every command without making any deviations, without going back with any loops to execute a command more than once. So in a straight line program, every command gets executed exactly once.

There is a very elegant, and even useful theory that talks about different layers of, levels of complexity of programs and says, for example, what kind of functions can you compute with straight line programs. We'll talk more about that field, which is called complexity theory, later in this semester.

But for now, the thing to realize is that straight line programs are just dead boring. You can't compute anything interesting with one. Last time we talked about a recipe as an analogy for a program. Imagine a recipe with no tests.

So every recipe, or almost every recipe I know, has some decisions. Taste it and add salt if you need it. Or poke at the meat and see if it's done. Or cook it until the thermometer says some degree on it.

Those are the kinds of tests we need to make interesting programs. The most primitive kind of test we see is what's called a conditional statement. And those are written using the word if, and optionally as we'll see, the words else or elif, standing for else, if.

So let's look at an example here. Where'd my mouse, oh there it is. Yes? Somebody has a question? Shout it out.

AUDIENCE: Sorry. I was wondering, when the user's prompted to put in the raw input, instead of putting in a float, puts in string, could you define it as a floating integer? How would you interpret that input?

PROFESSOR: I didn't get the question. So this is an argument to raw input, or their response to

raw input.

AUDIENCE: So yeah, for the raw input where you define it as a quote--

PROFESSOR: Yeah.

AUDIENCE: It usually puts in a string. How does Python interpret that?

PROFESSOR: It will interpret it as a string containing quotation marks.

AUDIENCE: OK.

PROFESSOR: So typically you don't type a string, because it interprets everything you type as if it were a string. So don't bother typing strings. Good question. Thank you.

All right. So let's look at this. So here I'm going to get an int, or at least a string. I'll convert it to an int. Then I'll say, if x remainder two, that's what the percent sign is, it's a remainder or a mod operator, is equal equal zero. That's important.

You'll notice that we used an equal sign to do assignments. If we want to do a comparison, whether two objects have the same value, we don't write a single equal. We write a double equal. So whenever you're testing for equality of objects, you use double equal.

So it says, if the object x mod 2 has the same value as the object zero, print even. Else, print odd. And then, just for fun, I'm going to see whether or not it's divisible by three.

Why did I do that? Just to show you that I can nest conditionals inside conditionals. So in one of the branches of the conditionals, I'm now doing a test. So what this does, is if comes down, it does the test.

If the value of the test is true, it executes the block of code following the if, in this case, just print. And then it skips the else. It does not execute the else. So it executes one or the other. If the test is false, it skips the block of code following the if and executes the block of code following the else. So it does a or b, but not both.

The indentation is important. Python is very unusual in that the way you indent things actually affects the meaning of them. And you can tell that, if I were to type this in the editor, you'll note here it's on that line, but if I hit Return, it automatically indents it. That's the auto indent feature I mentioned earlier in the editor of IDLE.

And this tells me how these things line up. So the fact that this is here tells me I execute it only as part of the else clause. The program would mean something quite different if I wrote this. Then it would mean, if x mod 2 is zero, print even. Otherwise, print odd. And whether or not it was even or odd, do this test in the if statement.

So the indentation actually affects the meaning of the program. Now a lot of other languages, almost all other languages, don't do that. They have some punctuation. For example, c uses set braces to designate what's called a block of code.

If you look, however, at a well-written piece of C code, or Java code, or any other language that I know, programmers are trained to use indentation to show the structure of the program. Even though you don't need, it you could line up everything right at the left edge and just use the punctuation.

People don't do that. And the reason they don't do that is programs are intended to be read, not just executed. Why are they intended to be read? Because the only reason, the only way you can debug a program is reading the code in it.

Typically, you want to write your program so that if you look at it from a distance, the visual structure of the program reflects the semantics of the program. And that's why people use indentation when they don't need to, so that you can see the structure of the program by looking at it on your screen and not having to parse each symbol.

The authors of Python made what I think is a very good design decision. They said, well, if that's the way you ought to write your programs, let's force people to write their programs that way and guarantee that the visual structure of the program actually matches the semantic structure.

The problem with languages like C and Java is that you can indent things and fool

the reader of the program by making it look like something is under something else, when in fact it really isn't, because of the punctuation.

So here we have a guarantee that the visual structure matches the semantic structure, and I think that was one of the really good design decisions in Python. OK, people see that?

So we could execute this program. Let me get back to what it was before. Control z is the go back. And now we can enter an integer, say 14, and it will tell us it's even.

I can run it again, and now I'll put 15 in, and it will tell me it's odd. We'll try it once more. We'll put in 17. It was odd and it's not divisible by three.

These kinds of programs are called branching programs. And that's because the structure of them, as you go down you execute some statements, and then there's a branch which says execute these statements or execute those statements. And then typically it comes back together and continues.

Of course, branches can have sub-branches. We could do this and then join further down, as we've seen here.

Now branching programs are much more interesting than straight line programs. We can do a lot of things with them, but fundamentally nothing really interesting. And we can think about that by thinking about how long it takes a branching program to run.

So let's first ask the question, how long does it take a straight line program to run? 14 seconds? No, that's not the way to think about it.

How would we think about how long it takes it to run? What governs the length of time a straight line program can take?

AUDIENCE: [INAUDIBLE].

PROFESSOR: Exactly. The number of statements or commands in the program. Since it executes every command exactly once, if you have 100 command, it will have 100 steps in it.

Now there's some variation on how long each step will be. Some commands might take longer than others, but the length of time it can take to run has nothing to do with its input. It has to do only with the number of lines of code. And that tells us it's not very useful because, well, we can only type so many lines in our lifetime.

Well branching programs have the same problem. In a branching program, each command is executed at most once. So again, the length of time it takes to execute the program is governed strictly by the size of the program.

Why isn't that good enough? Well, think about a program, say, to compute the GPA of all the students at MIT. Well how long is that going to take?

Think instead about a program to compute the GPA of all the students at the University of Michigan, which is probably 10 times bigger than MIT. Well you would expect that to take longer, right? Because you have to look at more students.

And in fact, it's true. Most programs that are interesting, the amount of time they take to run should depend not on the length of the program, but on the size of the data that you want to evaluate using the program.

So you would argue that the amount of time taken to compute the GPA of the students at MIT should be proportional to the number of students, not proportional to the length of the program used to do it. We'll talk a lot more about that later in the term in a much more thorough way. But it's important to get that as something you think about.

So the fact that branching programs are not proportional in time to the input means that they're limited in what they can do. So that gets us to the final concept we need to write every program that could ever be written, or at least to compute every function that could ever be computed. And that's some sort of a looping construct.

Once we add loops, we get to a class of programming languages or programming constructs that's called Turing Complete. And I mentioned this last time. Any program that can be written, or any function that can be computed, rather, can be

computed in a Turing Complete language.

So let's look at an example here. This concept, by the way, is called iteration. And if we look at languages with iteration, what we'll see is a more complicated flow of control. You execute some statements, maybe you do some branching if you want. But then you're allowed to go back and execute statements you've already executed.

Typically what you have is another branch. One branch goes back and one continues. So now we see we can execute a statement more than once. Suddenly we have enormous power at our disposal.

So let's look at an example of that. By the way, I'm skipping some of the code in your handout, but that's probably fine because it's there for you to be able to read. And what I would recommend by the way, is that we will post the handouts on the web, but at the end of every lecture within a few hours or a few days at least, go through the handouts and make sure you understand everything in. Because if you don't, you're probably missing something you'll need to understand to do the problem sets.

So here's a little program that finds the cube root of a perfect cube. This, by the way, is a useful comment here, right? Tells you what the program is intended to do.

So we get an integer. We set the variable ans to zero. And then while ans times ans times ans is less than the absolute value of x, we're going to set ans to ans plus 1. We could print where we are. I put those sort of things in as debugging statements.

If ans times ans times ans is not equal to the absolute value of x when I finish the loop, then I'll print x is not a perfect cube. Otherwise I have to do something to deal with positive and negative values.

Now I know that this was fast and that most of you probably don't fully assimilate this program. Do not worry. It will be discussed in recitations tomorrow. So tomorrow, the recitations will review the Python concepts we've discussed today, but we'll start by emphasizing how these loops work. OK. Thanks for coming. Enjoy

recitation tomorrow.